A MODULAR TESTBED FOR REALISTIC IMAGE SYNTHESIS

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Wayne Lytle

January 1989

CORNELL UNIVERSITY GRADUATE SCHOOL

APPROVAL OF THESIS

Name of Candidate Wayne (First Name) Thomas (Middle) LYTLE (Last)

Graduate Field ARCH. (Computer Graphics) Degree M.S.

Title of Thesis A Modular Testbed for Realistic Image Synthesis

Chairperson Michael F. Cohen Date Approved 10/6/88

Member [signature] Date Approved 9/20/88

Member [signature] Date Approved 9/20/88

Member _____ Date Approved _____

Abstract

A new Testbed for Image Synthesis is described. This Testbed was designed and implemented at Cornell University's Program of Computer Graphics to aid researchers in prototyping new rendering applications. It facilitates the development of new global illumination algorithms, and the testing of light reflection models. Tools are provided which reduce program development time and allow easy maintenance of the code comprising the Testbed.

This Testbed is compared and contrasted with some other testbeds. The overall structure is outlined, including the software library hierarchy and the contents and functionality of individual library modules. A standard interface to rendering software based on the Testbed is described. This interface allows compatibility with a variety of modelers, and has facilitated the development of image synthesis code independently from the modeling research efforts. The use of the Testbed is demonstrated in a ray tracing application which makes calls to routines in many Testbed libraries.

Biographical Sketch

The author was born in La Chapelle-St. Mesmin, France (in a U.S. military hospital) on Dec. 23, 1962. His family moved to Roselle Park, New Jersey, to Rome, Italy, to Wayne, New Jersey, to Rome again, to Princeton, New Jersey, to Sao Paulo, Brazil, and back to Princeton. At this point the author was a sophomore in Princeton High School.

After graduation, he attended West Chester University in Pennsylvania, receiving a B.S. in Liberal Studies. He concentrated in music (classical piano performance) and computer science. During this time he played synthesizer in a progressive rock band called Matrix Fulcrum, and later in a synthesizer duo named Sound Paintings. He was first introduced to computer graphics during his last two years at West Chester, and decided to pursue his interest further.

The author entered the Program of Computer Graphics at Cornell University in August 1986 to study for an M.S. degree. He continues his interest in music, composing and producing synthesizer music in his MIDI-based recording studio.

To Mom and Dad.

Acknowledgements

First of all, I would like to thank my parents for their love, wisdom, and support throughout the years, and for their patience despite the fact that I fell somewhat short of the model high school and college student.

I would like to thank Michael Cohen, my thesis chair, for the time he invested in advising me and editing my thesis, for his work on the design of the Testbed, and for being "one of us". Thanks especially for making the effort to fly back, after leaving Ithaca, for Stuart Feldman's and my defenses.

I would also like to thank Don Greenberg for giving me the opportunity to study in such an exiting atmosphere, and for the time he spent advising me, writing chapter 1, and editing my thesis. Special acknowledgement should be given to him for his initial foresight in starting the Testbed and his insights into its structural design.

Thanks also to my minor advisor (in Computer Science), Alberto Segre, from whom I took two very good Artificial Intelligence courses.

Many thanks go to Ben Trumbore (a fellow West Chester University graduate!), who played a major role in the overall design of the Testbed, and its implementation and maintenance. The second half of Chapter 3 is based on his writing for a soon

to be published paper on the Testbed. Thanks for the help with systems questions, and for lending me all those CD's.

I would like to thank Tim O'Connor, our resident UNIX guru and a fellow office-mate, for keeping the HP systems going, and for answering numerous systems questions.

Thanks to Stuart Feldman, the other Cohen-advisee, for stimulating conversations and psychiatric care, and for letting me use ray traced images of his Pavilion model in my thesis. Also thanks for the consulting on my Casino model. Special thanks go to Filippo Tampieri, for being my ray-tracing tutor and for answering many questions. Thanks to Rich Eaton, another office-mate, for being so cooperative in sharing the terminal.

Thanks to Jocelyn Avedisian for keeping me company at the lab when the hours got long.

Thanks also to Emil Ghinger for the photographic work. Special thanks go to Janet Brown-Aist and Ellen French for keeping the lab running smoothly.

I would like to acknowledge the following people for their Testbed library implementation work:

- Eric Chen (IBUF)
- Gene Dykes (TXS)
- Stuart Feldman (FED, FGP (destined to become FGN))
- Jim Ferwerda (CLR)

- Martin Livesey (GS)
- Kevin Novins (PAP (destined to become MFC), CQT, MQT, FED, FGP, SDB)
- Tim O'Connor (IBUF)
- Rod Recker (BV, RLE, SDB, RAD, ASA, FFC)
- Adam Stettner (XFM)
- Filippo Tampieri (FED, FGP)
- Ben Trumbore (MID, ATTR, NEWT, RIN, HBV, ANT)
- Michael Wilk (CLR)

(For those of you who were wondering, I worked on RAY, RIN, SRT, USS, ANT, and TXM, and wrote a ray tracer which makes calls to most of the libraries mentioned above.)

This research was supported in part by a grant from the National Science Foundation #CCR-8617880 entitled "Interactive Computer Graphics Input and Display Techniques". The implementation was conducted on equipment generously donated by Hewlett Packard Corporation.

Table of Contents

List of Figures

Chapter 1

Introduction

The goal of Realistic Image Synthesis is to generate images on computer displays which are virtually indistinguishable from a photograph. These images can be representations of existing or nonexistent environments. The image generation process requires an accurate representation of the physical propagation of light, a problem which represents an enormous computational task requiring sophisticated light reflection models. The current algorithms embedded in the workstation hardware being offered commercially cannot be extrapolated to handle these complex situations.

Algorithms which provide global illumination functions such as shading, shadows, color bleeding and interreflections, require new approaches for computer graphics. These algorithms must be accurate in the sense that they represent physical reality, but also become fast enough that the computational speeds are tractable. Furthermore, appropriate kernels of computation must be well defined so that future hardware implementations may accelerate these operations by sev-

eral orders of magnitude. To a large degree, the future hardware will also rely on parallel processing and pipeline architectures to provide the throughput necessary for interactive speeds.

For these reasons it was important to devise a Testbed to allow for the experimentation with light reflection models, the creating of global illumination algorithms, the parallelization of certain portions of the computer code, and the measurement of performance of different strategies. The Modular Testbed for Realistic Image Synthesis, developed at Cornell University's Program of Computer Graphics (PCG), tries to satisfy these goals. In particular, the Testbed has been structured to perform the following:

- Test new light reflection models and new global illumination algorithms so that experimental approaches can be implemented in a modular fashion.
- Measure performance characteristics by statistically monitoring subroutines calls and correlating the results with environment data to obtain predictive methods for computational cost.
- Render scenes and simulations of extremely high complexity, at least one to two orders of magnitude greater than what is currently being used.
- Provide an environment for the exploitation of coarse grained and fine grained parallelism for high level global illumination algorithms.
- Provide a mechanism to compare the computer simulations with actual measured physical results from laboratory testing.

- Reduce program development time by having appropriate modularity and interface descriptions so that experimental methods can be easily implemented.
- Provide a mechanism for easy maintenance of large portions of graphics software.

The system must be designed to not only fulfill these goals, but to comply with certain imposed constraints:

- Old modeling data be usable by new image synthesis algorithms.
- Data which is used for rendering must be independent of the modeling process, and cannot be restricted to a display list structure.
- The system must work on products from different manufacturers and be amenable to the rapid dynamic changes of the computer industry and graphics display technology.

The Modular Testbed for Realistic Image Synthesis is part of a global research plan and objective of Prof. Donald Greenberg's grant from the National Science Foundation entitled "Computer Graphics Input and Display Techniques". It represents the collective effort of many persons at the Program of Computer Graphics. The overall design was the result of the planning efforts of faculty, staff, and students. The implementation and testing of the library modules was carried out by several students and staff members, mentioned by name in the Acknowledgements.

This thesis presents the design, contents, and uses of the Testbed. Chapter 2 provides a historical perspective on image synthesis research, including a discussion

of some other testbeds. Chapter 3 describes the overall design and structure of the Testbed, while Chapter 4 details the contents of individual libraries. The implementation of an application using the Testbed, a ray tracer, is outlined in Chapter 5. The final chapter contains observations concerning the successes of this software endeavor, and discusses the extent to which the goals have been met.

Chapter 2

Historical Context

2.1 An Overview of Realistic Image Synthesis

In the real world, electromagnetic radiation propagates through the environment in predictable ways, following well understood physical laws. As this light enters our eyes, a retinal image is formed, and information about this image propagates to our brain. It is well understood that perspective illusions may be created by generating an image on a picture plane, resulting in the perception of a scene.

Systems used to generate computer images involve the implementation of mathematical formulations describing this process for light propagation in the visible range. The image formed consists of a set of sample intensities which represent all visible light passing through the image plane and reaching the eye. This image is viewed on a computer display comprised of discrete picture elements (pixels), each capable of displaying one of a discrete number of colors.

Realistic Image Synthesis can be described as a four stage process [HALL83], where the stages may be completely separate, or may be integrated in various

ways. For example, Ray Tracing combines the second and third stages.

The first stage is the modeling of the environment. This involves the specification of the geometry, position, and orientation of each object. Attributes are assigned to objects describing surface color and reflectance properties, transmittance, texture, and various parameters needed in the rendering process. Also, the environment's lighting is described.

The second stage is concerned with determining the global illumination for the environment, which is a view independent process. All light arriving at each surface is the summation of all lighting arriving *directly* from the light sources and all light arriving *indirectly* (through reflections from other surfaces or transmittance through the given surface). The light leaving the surface in a certain direction is determined from the above data and the physical surface properties.

Not all global illumination data is useful or necessary for the calculation of the final image, and some rendering processes focus exclusively on those portions that directly pertain to the given view of the environment. This is implemented in the view dependent third stage where, for each pixel, the intensity at the eye (arriving from the environment through the viewing plane), is determined for a specified viewing position, direction, and frustum. This includes the determination of the surfaces visible from the eye point.

The final stage in the image generation process involves the conversion of the intensities at the image plane to a displayable form. By keeping this stage separate from the previous stage, the image plane data can be manipulated in various ways, transformed to different display resolutions, and adapted to the characteristics of

the display device and/or photographic equipment.

2.2 A Brief History of Rendering Techniques

Early computer graphics images were line-drawings generated on vector displays. Advanced systems performed perspective transformations, hidden line removal, and depth cuing. Objects were displayed as a collection of edges, each rendered as a line between two points on the screen. Most objects displayed were polygonal, and even curved surfaces were represented by their polygonal approximations. Basic transformations, clipping to the view window, and depth cueing were implemented in hardware, resulting in very fast interactive display environments. Vector display systems are still in use today to some extent, especially for CAD and other modeling applications, but are giving way to raster display systems.

A raster display device is comprised of a color monitor coupled with a frame buffer. These devices are capable of displaying one of a finite number of colors (currently the standard is over 16 million), at one of a finite number of screen locations (typically 1024 scanlines of 1280 pixels).

Historically, the steps for generating a color image has been as follows. First, a perspective transformation is applied to all objects in the environment, based on the view. Next, objects outside the viewing frustum are clipped and discarded, and a visible surface algorithm is then applied. Lastly, surface color is determined through the application of a reflectance model.

While the first two steps are very simple, the hidden surface removal stage presents more of a problem, and was in fact a very active area of graphics research

for a number of years. Many solutions to the hidden surface problem now exist, and can be found in the literature [SUTH74]. Hidden surface techniques can usually be classified as either object space or image space algorithms. The former performs hidden surface calculations at the precision of the environment, while the latter performs a very efficient visible surface calculation taking advantage of pixel to pixel and scanline to scanline coherence[WARN69] [WATK70].

It is interesting to look at the development of techniques to apply reflectance models, because several of the techniques developed have been integrated into more advanced rendering algorithms, and have been implemented in hardware by some manufacturers. Initially, object color was simply based on diffuse Lambertian shading. The intensity was simply a function of the angle between the light source and the surface normal. This accounted for the reflected intensity being equally distributed in all directions upon leaving the object surface.

Gouraud suggested interpolation of intensities across polygonal objects to approximate curved surfaces [GOUR71]. Phong improved on this by interpolating normals across polygon faces, and applying the reflectance model at each pixel. He also added a specular contribution to the reflectance model to account for highlights [PHON75]. This was dependent on the viewer position, incident light direction, and surface reflectance properties.

Blinn made further improvements by incorporating part of the Torrance-Sparrow lighting model [BLIN77]. Blinn also improved the specular component for mirror-like surfaces, and proposed various methods for mapping textures onto object surfaces, greatly increasing the perceived complexity of the image.

As these reflectance models evolved, and the rendering methods were enhanced, the degree of realism increased. However, the shortcoming of the process outlined above is that only "local" lighting affects are accounted for. Missing are shadows, indirect illumination from reflection, refraction, and other features that give the viewer more visual cues about the environment. Although later algorithms emerged that provided shadow generation [CROW78] [ATHE78], these solutions were still not global, since surface inter-reflections were ignored.

As a solution to the global illumination problem, a very promising method emerged, called Ray Tracing. Ray casting was originally suggested by Appel to solve the hidden surface problem [APPE68], and was implemented by Goldstein and Nagel [GOLD71]. In his classic paper, Whitted presented the Ray Tracing algorithm as a tractable means of determining the global illumination of the environment from a certain view [WHIT80]. The process consists of tracing rays from the eye position, through each pixel on the image plane, into the environment. The first surface struck by the ray is determined, and a new ray is spawned in the reflection direction for specular objects, and in the refraction direction if the object is transparent. This process is repeated for these new rays, and continues recursively forming a ray tree.

At each tree node (representing an intersection point), a lighting model is applied to determine the intensity contribution at that point. This is added to the contribution from reflected and refracted rays farther down the tree, and is returned as the intensity for the given pixel. The calculation of the intensity at each node requires determining whether or not the point is in shadow for each light

source. This involves shooting rays in the direction of each light and testing for intersection with any opaque object. If the surface point is "seen" by the light, then the contribution from the given light is added to the total contribution for that node.

Images rendered with the Ray Tracing method have produced some very impressive and realistic images, but these images are not completely accurate for several reasons. First of all, the reflectance model used is empirical. The theoretical behavior of light is not accounted for, and samples are usually only taken in the directions of mirror reflection and transmission. The method fails to account for the global effects of diffuse reflectance within the environment. Also, the use of point sampling results in undesirable aliasing artifacts. Many enhancements on the original Ray Tracing algorithm have been made, and research continues to produce improvements in both image quality and rendering time.

In 1984, a new rendering approach called Radiosity was introduced, based on concepts from thermal engineering [GORA84] [COHE85b]. A truly global solution to the illumination problem, it models the full interaction of light between Lambertian diffuse surfaces, and includes features such as soft shadows and color bleeding. The intensity at a given point is no longer just a function of the object surface and light, but of the whole environment. Radiosity is a view independent method, and actually occurs in the opposite order from traditional techniques. Instead of establishing viewer position and direction, and then calculating intensities for all visible surfaces, the surface intensities for the entire environment all calculated first. Then view parameters are selected and the final image is rendered.

Inter-surface relationships are defined through a set of linear equations describing the light energy balance between reflection and absorption of light emitted into the environment. The solution of these equations constitute the simultaneous global solution for the intensity of light leaving all surfaces. This results in the increased realism of shading and interreflection of diffuse surfaces. Once the surface intensities are determined, rendering proceeds using any standard hidden surface technique, including very fast methods implemented in hardware. Since the computation is view independent, generating multiple views for dynamic sequences (for example, environment "walk-through's") are quite inexpensive. Also, most of the computation is independent of material properties, making changes in surface colors and/or lighting conditions very fast.

2.3 Recent Research in Realistic Image Synthesis

Today, research in Realistic Image Synthesis covers a very wide range of areas. Most research topics can be characterized as being in one of two categories: decreasing rendering time or increasing image realism. The Program of Computer Graphics is involved with both aspects of the research, and has pioneered many of the developments. The Testbed has been developed for the purpose of assisting in this research.

2.3.1 Decreasing Rendering Time

Ray Tracing is a very computationally intensive process, and it is desirable to minimize expensive calculations such as ray-object intersections, and shadow testing. In the basic Ray Tracing algorithm, the process of determining the closest object hit along a given ray involves performing the ray-object intersection test with every object in the environment. Several methods are described in the literature which reduce the number of ray-object intersection tests. Rubin and Whitted explored the use of hierarchical bounding boxes [RUBI80]. Bounding boxes were placed around individual objects and collections of other bounding boxes. Relatively inexpensive ray intersection tests were performed with the bounding boxes first, and only if the ray intersected the bounding box were the intersection tests performed with its contents. Hooper and Weghorst explored ways to place bounding volumes around complex objects, and found that different object types often had different optimal bounding volume types [WEGH84].

Hooper and Weghorst devised a method involving a visible surface preprocess, where all objects are scan-converted into a variation Z-buffer variation called an Item Buffer [WEGH84]. Instead of a color value, as in the case of the Z-buffer, each entry contains the object and surface ID of the closest hit along the ray through the given pixel. For first level rays, those originating from the eye, the process of determining the closest hit involves an Item Buffer table lookup to determine the closest object and surface, and then just a single ray-surface intersection.

Another method for acceleration of Ray Tracing takes into consideration the fact that a very large percentage of the total rendering time is spent determining

whether a given intersection point is illuminated by a light source. For each light, this would typically involve performing a ray intersection test with each opaque object, until either an object is found that shadows the point from the light, or the object list has been exhausted. In the latter case, a lighting model would then be applied to calculate the intensity contribution from the given light. Haines proposed a method to partition the environment with respect to each light by creating a Light Buffer [HAIN86], where each entry contains a list of objects which potentially occlude the light for that discrete area on the buffer. These lists are short, so the number of objects to be tested for a given point is small, and the total shadow testing time is greatly reduced.

Glassner described a way to recursively subdivide space, using an octree technique [GLAS84]. The environment is subdivided into orthogonal non-overlapping cubes of various sizes, each containing no more than a specified number of objects. Each ray efficiently moves from cube to cube until a ray-object intersection is found. A similar technique was implemented by Kaplan, involving Binary Space Partition trees [KAPL85]. Non-leaf nodes contain slicing planes which are aligned with two axes and each divide space into two infinite subspaces. Leaf nodes consist of a description of the cubic subspace defined, and a list of contained objects. Finding the closest hit involves traversing the BSP tree to reach the appropriate leaf node containing an object list. The ray is intersected with only these objects, and the closest hit returned.

Dippe and Swensen proposed a way to adaptively subdivide the environment into subregions of approximate uniform complexity [DIPP84]. These subregions

were mapped onto nodes in a parallel environment, resulting in roughly uniform load distribution. Arvo and Kirk described a unique approach to fast ray tracing involving 5D space subdivision [ARVO87]. Rays are classified into 5D hypercubes, by examining ray origin and direction. This method exploits object coherence and image coherence, drastically reducing the number of ray-object and ray-bounding volume intersections.

2.3.2 Increasing Image Realism

A significant amount of research has been done involving the increased realism of generated images. This research usually explored better methods to model the behavior of light, both its behavior at object surfaces, and its propagation through the environment.

As mentioned earlier, Blinn introduced an improved lighting model [BLIN77] based on research and light reflection measurements by Torrance and Sparrow [TORR67]. This model contains a more accurate function for generating specular highlight, and differs from the Phong model in that the magnitude of intensity, spatial distribution, and position of specular highlight vary with the angle of incidence. It is based on the assumption that surfaces are composed of microfacets. The specular reflection distribution is a function of the angle between the mirror reflection direction and the surface normal, and is dependent on the microfacet distribution, amount of microfacet shadowing and masking, and the index of refraction.

Cook and Torrance presented an energy based lighting model [COOK81], which

handles rough surfaces in a more general way. This model correctly produces directional distribution and spectral composition of reflected light, accounting for the color shift in reflected light as the angle of incidence changes. Wavelength dependent spectral energy distributions and reflectances are used. Specular highlights are a function of surface material property. Fresnel equations are used to predict the spectral composition as a function of wavelength and angle of incidence. As the grazing angle (90 degrees) is approached, the composition of the reflected light approaches the color of the light source.

In an effort to account for soft shadows, blurry reflections, translucency, and other fuzzy phenomena, a stochastic sampling technique was incorporated into the Ray Tracing process, resulting in Distributed Ray Tracing [COOK84]. This Monte Carlo method is used to evaluate the integrals over pixel area, and reflected and refracted hemispheres. The rays are distributed over the specular distribution function, the transmitted directions, the solid angle of each light source, etc.

Although this increased the realism of Ray Traced images, it is still an approximate solution which fails to account for certain phenomena, such as caustics and color bleeding. In order to correctly account for all light leaving a surface, all incoming light from all sources (lights, and all other visible surfaces) must be considered. For this to be done, it is necessary to solve the rendering equation [KAJI86]. Two approaches were presented in 1986, which differ mainly in their choice of what to discretize. Kajiya proposed a Ray Tracing method using a Monte Carlo solution, where the image plane was discretized. Immel [IMME86] suggested a Radiosity based method, where the environment was discretized.

It is necessary to account for the light arriving at a point on an object surface coming from all directions. In order to do this, sampling must be done over the whole reflectance and transmittance distributions. In Kajiya's method, at each surface intersection, a ray is shot at the light and in a randomly chosen direction based on the reflected and transmitted distributions [KAJI86]. Rushmeier enhanced this by taking into account area light sources [RUSH87].

Two methods that have emerged that combine Ray Tracing and Radiosity techniques. Wallace observed that view dependent methods such as Ray Tracing were good for calculating the specular component of light reflection, while Radiosity, being a view independent method, provided an accurate description of diffuse reflectance phenomena [WALL87]. A hybrid approach was suggested, involving two separate passes. First, a view independent preprocess is performed, based on the hemi-cube Radiosity algorithm [COHE85b], which is extended to include diffuse transmission and specular to diffuse reflectance and transmission. Then, the view dependent stage is executed, which involves a variation on distributed ray tracing. A small Z-buffer is used to sample intensities contributing to specularly reflected or transmitted intensities.

Rushmeier explored the possibilities of rendering environments which include participating media, such as fog, smoke, and clouds [RUSH88]. Two avenues of approach were taken. The first uses radiosity-based enclosure theory extended to include both isotropically and anisotropically scattering media. The second approach uses Ray Tracing methods to calculate the radiant energy exchange within an environment. This is derived from general equations of radiative transport,

and is implemented using Monte Carlo techniques. The two approaches are combined in another hybrid of Radiosity and Ray Tracing, where a Radiosity-based preprocess precedes the Monte Carlo stage.

To realistically represent light at a given point in an environment, a renderer must account for light arriving at that point from all directions. All lighting models approximate this by simplifying the rendering equation and focusing on only some of the contributing components. There are many ways to discretize this continuous phenomenon. It is necessary to find tractable lighting models which provide satisfactory results. The Testbed is a vehicle for exploring new lighting models.

2.4 Past and Current Testbed Systems

In order to facilitate research in computer graphics, various universities and other research centers have developed testbed systems. These take on many different forms and are geared toward a variety of areas of research, including modeling, rendering, and animation. A series of questions can be asked about a given testbed system.

- What is its objective in computer graphics research?
- What constitutes a component, and what is the granularity of the system?
- How are the components interconnected?
- Is the system designed to grow in supported object types or functionality (or both or neither)?

- Is the testbed comprised of a set of building blocks which can be interconnected to construct various experimental systems, or is it a complete system itself possessing interchangeable or flexible parts?

These questions are answered in different ways for each testbed. Several testbed systems, which pertain mainly to rendering areas are described below.

One of the first testbeds appearing in the literature is a system developed at Bell Laboratories in 1982, which was used for scanline rendering [WHIT82]. The rendering algorithm used consisted of a three component pipeline: transformation/clipping, scan conversion, and shading. This testbed facilitated the construction of renderers which allowed the simultaneous processing of several object types in a single pass. An assortment of commonly needed routines was provided. These routines could be assembled in various ways. The goal of this system was to minimize the amount of special programming required for each new rendering application, and the building block approach contributed greatly to the accomplishment of this goal.

Each element was a separately compiled subroutine, written in C. The assemblage of these elements involved writing a program consisting mainly of calls to these routines. Communications between different elements where routed through a structure common to all, called a "span buffer", which carried data pertaining to the 3D object while having some characteristics of a run length encoded image. Two separate shader building blocks were provided, implementing the Gouraud and Phong models respectively, which could easily be interchanged. The existence of a collection of commonly needed rendering utilities allowed rapid prototyping

of specialized renderers.

The original Testbed for Image Synthesis implemented at Cornell was designed to aid in research mostly pertaining to aspects of the Ray Tracing rendering process [HALL83]. This included lighting models, the rendering of parametric surfaces, reflections, light propagation, and texture mapping. The form this testbed system took was a series of global "resource manipulation programs", where resources included transformations, objects, lights, materials, texture maps, and reflection models. These programs communicated through files, and were connected in a rigidly defined way. Included in the system were programs for editing various types of geometric objects such as quadric surfaces, parametric surfaces, and polygonal objects. A separate program was used to assemble these objects in an environment. Programs were provided to assign various rendering parameters, and to perform pre-rendering tests on isolated parts of the image to be generated. Finally, the actual Ray Tracer renderer, and image manipulation and display programs were provided.

The system was implemented in a pseudo object oriented fashion, and included a rich collection of object types. Each object was required to perform a specific set of operations including object creation, reading from file, writing to file, local to global coordinate transformation, wireframe drawing, and ray intersection calculations. This methodology of modularization allowed easy addition and testing of experimental object types, and the availability of testing programs made the debugging of new system components a smooth process. Two things should be stressed when analyzing this system. First, the object oriented approach is geared

toward experimentation with new object types, and not with different rendering functionality. Second, this was one completely integrated system designed to perform one type of rendering, namely Ray Tracing, in a flexible fashion. These were at least implicitly part of the design, and the system proved to be successful in meeting these goals.

Another testbed system was developed at Bell Laboratories much more recently by Potmesil and Hoffert, and was named FRAMES (Flexible Rendering, Animation and Modeling Experimentation System) [POTM87]. This system also was used to construct image rendering pipelines, but in a very flexible manner. The basic building blocks are small programs, UNIX filters, which are interconnected using the UNIX piping facility. These are "loosely coupled" in the sense that they are not compiled together in some mainline program, but are combined at runtime by a single command line description. Only those filters needed for a given application are invoked. As new techniques evolve and new tools are developed, they are added to the repertoire of tools, allowing controlled and flexible growth. The system is ideal for experimentation with distributed rendering, as a means is provided to assign each filter to a separate processor on a network. It must be noted, however, that scanline rendering methods are particularly suitable for implementation as pipeline systems, while global illumination algorithms, by their very nature, cannot usually be executed in a simple sequential pipeline fashion.

In the same year, a testbed system based on a similar notion of "loosely coupled" nodes was presented by Nadas and Fournier, called GRAPE (General Rendering Applications Programming Environment). In [NADA87], the authors com-

ment that while previous testbeds allowed experimentation with new object types and provided flexible ways to interchange modules, the ability to experiment with the very structure of the system was lacking. Based on data-flow notions, the GRAPE system is more flexible than FRAMES in that the topology of the system is not limited to a linear flow. Arbitrary acyclic graphics can be assembled, allowing nodes to have more than one input or output. A stated goal is to allow flow of control experimentation entailing little or no coding.

The basic building block is a node, which is written in C. All nodes input and output the same flexible structure called an *appel.* As data flows through the system, the appels metamorphose from modeling objects such as parametric surfaces to pixels at the final output node. The connection is performed using UNIX directory manipulation tools. The programming environment exists on two levels, each geared toward potentially different users. The *macro* level involves node assembly, which is performed by the designer of the experimental system. The *micro* level consists of actual node implementation and is carried out by C programmers.

The authors claim that because this system is not slanted toward any particular rendering technique, like so many other testbeds are, it facilitates the rapid and flexible experimentation with new ideas. However, it may be suggested that global lighting affects are still out of reach by this system. Perhaps the restriction that node assemblies may not contain cycles is the culprit, or it may be that the interconnected node methodology itself is not able to elegantly handle the global illumination problem.

At the opposite extreme concerning system granularity, is the REYES image rendering system, developed at Lucasfilm and currently in use at PIXAR [COOK87]. Designed more as a production system than a testbed, this system strives to efficiently render environments of very high model and shading complexity. REYES is another example of a "monolithic" system, which is geared toward one specific rendering technique, although a "back door" is provided to allow the incorporation of other existing or new rendering methods for some objects in the environment. Though the system has been used with a high degree of success for its intended purpose, the flexibility of the interface for other rendering methods has not yet been clearly demonstrated.

Most testbed systems can be characterized as falling into one of three classifications: monolithic systems with many run time options and possibly "back door" interfaces, assemblies of relatively simple nodes linked together through UNIX pipes or similar mechanisms, or collections of isolated specialized tools (in the form of software libraries) called by higher level user-written programs. The Modular Testbed for Realistic Image Synthesis, described in this thesis, fits the latter classification.

Chapter 3

Overview of the Testbed

In this chapter, the main ideas behind the design of the Testbed software are outlined and the usefulness of the Testbed in building experimental applications is highlighted. The organization of the Testbed software is described, followed by some comments on implementation approaches.

3.1 General Discussion

3.1.1 The Advantage of Having a Testbed

The two main steps involved in computer graphics research are the development of ideas and the implementation of software to demonstrate the validity of these ideas. A very large percentage of the time spent doing research is dedicated to software development. It is conceivable that similar or perhaps even better results could be achieved if less of the total research time was spent writing code, and more time was spent developing ideas.

In the past, the development stage of experimental graphics software was not

very structured. An application would typically be developed by either one person or a team of two. The developer or developers would either begin implementation from the ground up, or would search for some other available software which had a similar function to that which they were attempting to create. If this search was successful, the developers would then "borrow" the code. This would consist of copying the original code, then modifying it to perform the desired function.

The problem with the first approach is code duplication. If several developers write code performing nearly identical tasks, this represents a serious inefficiency for a research team as a whole. The second approach has two inherent disadvantages. First, since presumably the data structures in the new application are not the same as those in the original, it is necessary to go through the entire code modifying it to reflect these differences. Second, any updates made to the original code, either in efficiency or function, are unavailable to the developer who "borrowed" the code, apart from beginning the the process again.

The Testbed provides an alternative to these approaches by providing a means of avoiding code duplication and allowing the work of a single individual to be useful to many others. As research problems and their implementations grow more complex, it becomes increasingly difficult to write entire applications in isolation from others. It has become necessary to make use of the software developed by other members of the research community, and the Testbed provides a mechanism for doing this.

3.1.2 Definition of the Testbed

What is the Testbed?

The Testbed for Image Synthesis is a collection of various tools provided to researchers. It was designed to aid in the development of experimental rendering applications, and will be used to test new lighting models and image synthesis algorithms.

The Testbed is comprised of several interrelated components. There is a three-level hierarchy of software libraries which provide most functions commonly required by a typical rendering application. A description of this framework is given in Section 3.2.2, while the individual library functionality is detailed in Chapter 4.

Several file formats are defined. These allow a common description of files used by all rendering applications. Included are the formats of both the input and output of the rendering process. Object environments are stored in the Modeler Independent Description format (MID), described in Section 3.2, and raster images are stored in run length encoded format (RLE).

Runtime structures are another part of the Testbed which unifies the code of developers. This guarantees that developers with similar program functionality will be using identical structures, since they will be including the same structure definitions that are provided with the libraries to which they are binding. Definitions of these structures are usually found at the utility library level. For example, anyone using rays in their application will bind with the RAY library, within which the RAY data structure is defined.

Test data is also a component of the Testbed. To accelerate the development

and testing process, standard files are provided for use as input to the module or application being tested. For example, the test environment directory contains environments of various complexity from a single object to many thousands of objects.

Collectively, these components form a mechanism that allows the sharing of code by many graphics applications programmers, and provide a way to avoid excessive duplication of effort.

What is the Testbed NOT?

The Testbed can also be characterized by its dissimilarity to various other graphics systems.

First and foremost, the Testbed is not a production system. The purpose of the Testbed is very different from that of a production system in several ways. In a production environment, the focus is on the product: the final image. Our focus is more on the rendering process itself. The goal described here is the generation of accurate realistic images through sound scientific formulations, as opposed to the production of images for customers in the advertisement or entertainment industries.

Two very important features of the Testbed are generality and flexibility, provided sometimes at the expense of program data space and execution time. This is diametrically opposed to the typical special purpose rendering programs used at production houses which produce specialized images at high efficiency. A very important goal of the Testbed is to provide clean code with very straight forward

interfaces. Rendering speed is attained by the use of efficient algorithms, the minimization of function calls, and the use of parallelism, as opposed to clever "hacks" or machine dependent optimizations to gain speed at the functional level.

Quick construction of test programs is an important use of the Testbed, and highlights a major difference between this process and the development of software sold by graphics vendors. In general, the test programs exist to demonstrate a specific concept. Testing and program use merge together, while programs are expected to have limited usefulness and a relatively short life. On the other hand graphics systems sold by vendors tend to be very robust, well tested and reliable, and the software development effort reflects this.

The Testbed described also differs from several existing testbed systems that have been developed at other research facilities. The difference in this case tends to be less one of purpose and more one of structure. The building blocks in the Testbed are software libraries which are bound into applications at link time. This is different from the technique used in several other testbeds where building blocks are actually mini programs that are linked together by UNIX pipes.

Our Testbed system is not a massive renderer with flexibility provided in the form of software "back doors". Instead it is a means to build various forms of renderers, and does not cater to any one specific rendering technique.

Finally, this Testbed is not an object oriented system existing for the purpose of experimenting with different object types. The focus is on image synthesis techniques. Although several object types are supported, the emphasis is on experimentation with different global illumination models, as opposed to complex

object types.

3.1.3 Software Requirements

The Testbed software must be logically organized into clear modules. This modularity is accomplished through the library framework which groups together routines of a specified functionality. Individual modules must not have excessively large scope, but should specialize in one area of rendering. Generality should be achieved by providing a moderate collection of simple yet flexible functions, as opposed to large, general functions exhaustively allowing alternatives. Their structure should be such that maintenance, even by other developers, is not overly burdensome.

The structure of the components and functionality of routines must cater to research goals. Two important areas are high complexity and parallelism. Functions and data structures must be able to handle extremely large environments, and must function acceptably in a course grained parallel environment.

Finally, all components of the Testbed must be flexible and extensible. Code for the libraries must be written in a fashion that lends itself to change. The changes should be simple to make at the library level and as transparent as possible to the applications programmer using a given library. No decision or rule should prohibit extensibility of the Testbed. It is necessary and desirable at times to prescribe constraints within which to work, however these must not be limiting in any way to possible future directions of the Testbed.

3.2 Organization of the Testbed

3.2.1 The Testbed's Role in the Rendering Process

Ideally, environments generated by any modeler should be compatible with any renderer. Also, the rendering process should be independent of the modeling process. Furthermore, the development of rendering software should be isolated from the development of modeling software. This constraint helps define the boundaries of this Testbed, which pertains to rendering applications only.

To meet the above requirements, all renderers must produce images from a single environment description. To resolve this need for unlimited modeling data formats and a single rendering data format, the Modeler Independent Description (MID) data format was defined.

MID serves as the interface between modeling programs and rendering programs. Figure 3.1 depicts the high level structure of the Testbed. The modeling programs, on the left, read and write their own private data formats, and may use any available display structure to produce the interactive graphics communication used during modeling. Several modelers may share the same private data format, or all formats may be unique. These data formats can all be converted into MID, but MID files cannot be converted back to the modeler data formats. Generally, the conversion of modeling data to MID is only used when a high quality rendering is to be produced.

A single software module is used by all renderers to read MID data and create standard data structures. Renderers may use the information in these standard

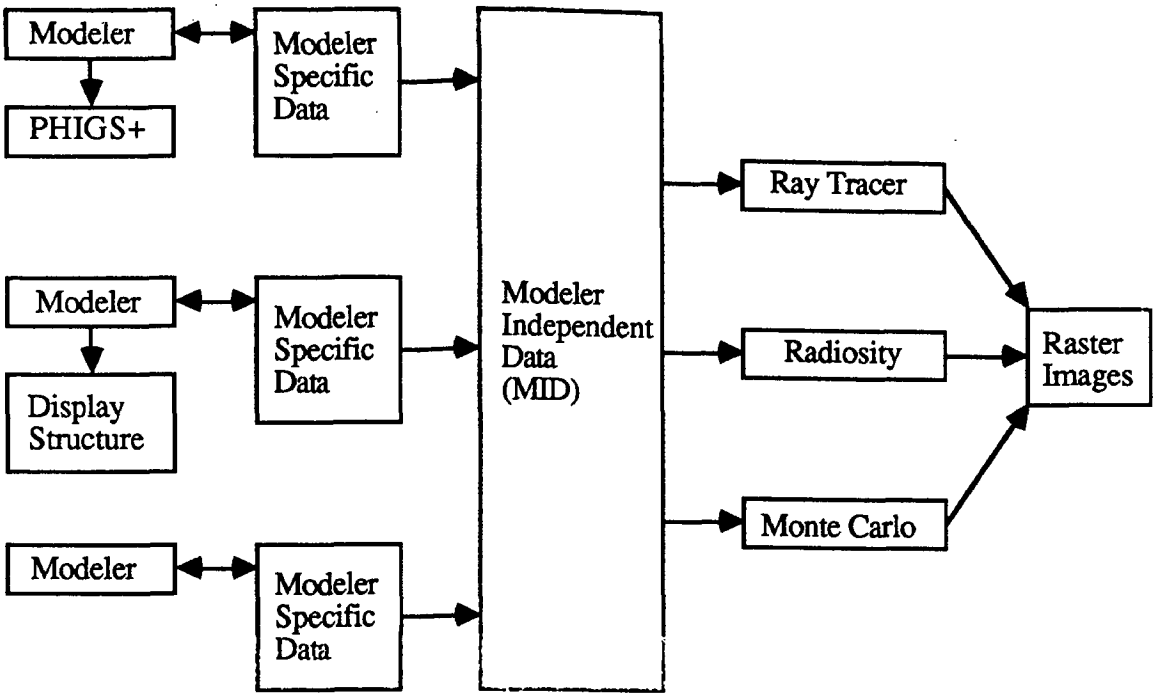

Figure 3.1: High-level structure of the Testbed

structures to construct their own local data structures. The local structures used by a Ray Tracer, a Radiosity renderer, and a Monte Carlo renderer are likely to be substantially different. The rendering programs themselves are constructed using functionality from a variety of Testbed modules. Each of these renderers produces raster image files in a standard format.

3.2.2 The Modeler Independent Description (MID)

MID is a text format that describes a list of primitive objects and their attributes. Because the hierarchies used for interactive display are not usually appropriate for renderers, no display structure hierarchy is described by MID. However, MID

does allow trees of primitives to be formed using the boolean set operations *union*, *intersection*, and *difference*.

Because the Testbed must support environments of unlimited size, routines that interpret the MID format must be able to read objects sequentially. If an environment is larger than a computer's virtual memory, it cannot be read all at once. If some objects in the environment instance the transformations and attributes of other objects in the environment, then the entire environment would need to be retained in memory. For this reason, MID does not allow instancing of transformed objects. However, because some object geometries are defined by a large amount of data, several objects may reference the same geometric data (which is stored separately from MID).

Each object in a MID environment is defined as a primitive type and a limitless list of attributes. These attributes are specified by *name-value pairs*, in which an attribute is named and its value is specified. An attribute value can be a scalar number, a text string, or the name of and directions to find a data block stored separately from MID. Three attributes are predefined by the Testbed: *transformations, geometric data,* and *rendering information.* Transformation attributes define the way a primitive object is transformed from object space to world space. Geometric data attributes complete the physical description of those primitive objects that require additional parameters. For example, the definition of a torus depends on the lengths of its major and minor radii, and a polygonal object requires list of vertices and polygons to define its shape. Rendering information includes material, surface, and emission properties.

New image synthesis algorithms using the Testbed may define other attributes as they are needed. A renderer can be written to look for certain attributes, interpreting their values as it sees fit. Renderers that do not use a given attribute can simply ignore it. This open-ended attribute system provides the basis for consistency between renderers, but does not hinder future research.

Because the MID format is simple yet extensible, it allows old models to be used by new renderers, and new models to be used by old renderers.

3.2.3 The Module Hierarchy

The Testbed's modules are organized into three conceptual levels of functionality (Figure 3.2). Modules at a given level may use other modules at the same level, or any level below it. Image synthesis algorithms built upon the Testbed may use modules from any level.

The lowest level of the Testbed contains **Utility** modules. These modules provide the most basic functions. Some Utility modules create and manipulate data structures, such as those for environments, attributes, polygons, and raster images. Other Utility modules perform low-level mathematical functions, such as transformation matrix operations and the solution of sets of simultaneous equations.

The middle level of the Testbed contains **Object** modules. Each object module performs certain functions for all classes of objects in the Testbed. When a new object class is added to the Testbed, functionality for this object class must be added to each Object level module. If a new Object level module is added to the Testbed, functionality must be included in that module for each object class in

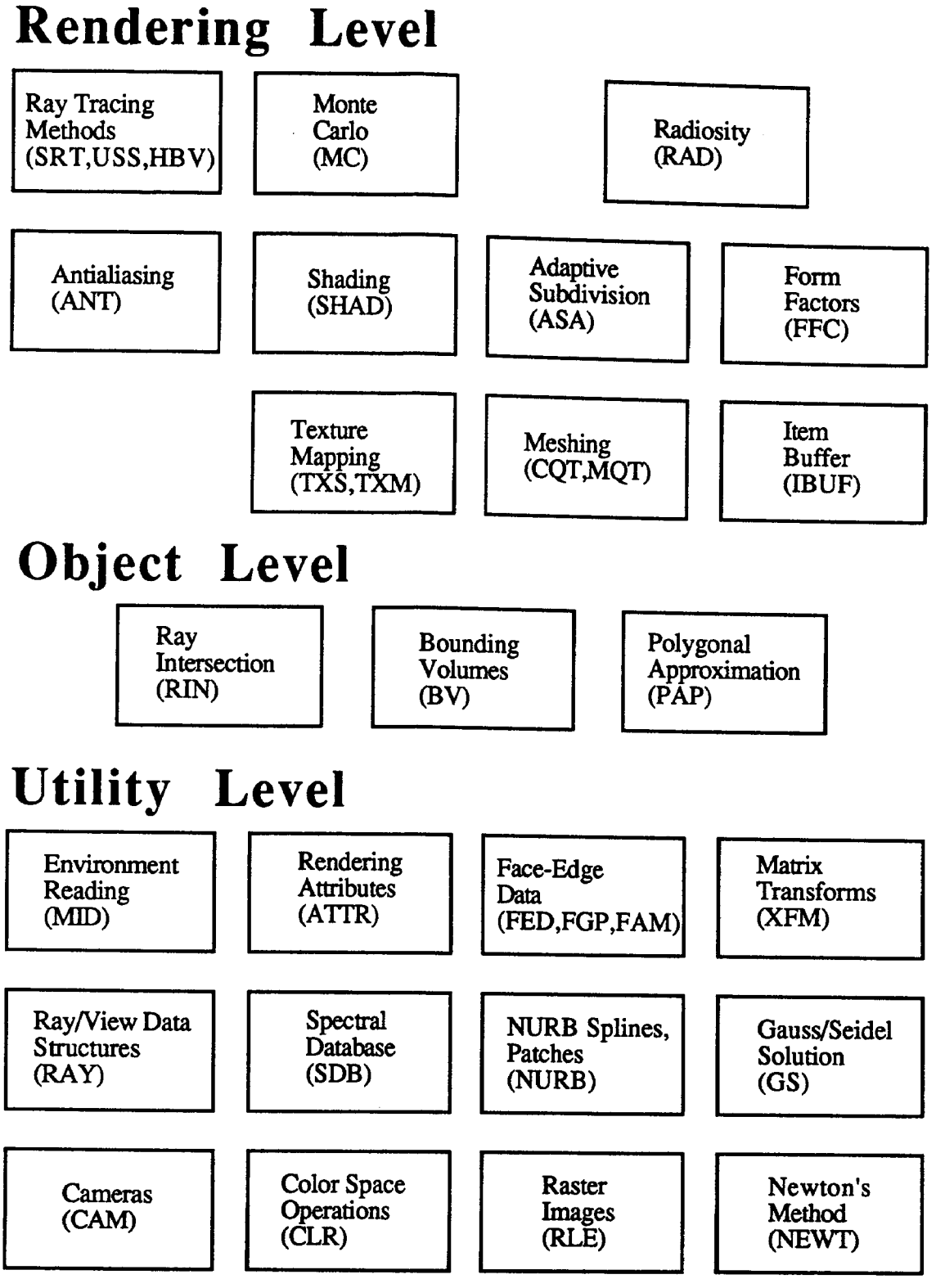

Figure 3.2: Three-level structure of Testbed modules

the Testbed. Object level modules include intersecting rays with objects, creating bounding volumes for objects, and approximating objects with a collection of polygons. These modules allow applications and other libraries to treat all objects the same, regardless of their class.

The highest Testbed level contains **Rendering** modules. These modules provide simple interfaces for complex rendering techniques. They perform such rendering tasks as shading, texturing, radiosity calculation, and the use of hierarchical structures to make ray tracing more efficient. Researchers can easily implement experimental rendering algorithms by using several modules. Individual modules can then be replaced with specific software the researcher has written. This minimizes development overhead and allows the researcher to concentrate on new functionality.

3.3 Implementation Approach

Several things must be considered when approaching the task of implementing software for a Testbed like the one described in this thesis.

Some libraries will have to be designed and implemented completely while other libraries are still just sketched ideas. The implications of this incremental development include the necessity of isolating functionality to minimize interdependence. When dependencies do exist, care must be taken to ensure design stability at the lower level before beginning the higher level library. Also, since the proposal of a library is often based on immediate need, the developer must avoid the tendency to specify functionality based on limited vision, i.e. only what is needed by the

current application.

Interfaces provided to these libraries must be powerful yet versatile. In order to fulfill both these requirements, it is sometimes necessary to provide more than one level of interface. At the highest level, a single call will provide the maximum functionality given a certain set of parameters. At a lower level, more specific subroutines will perform simpler functions given a subset of the parameters.

Care must be taken to provide very general library interfaces, maximizing the usefulness of their functionality. In order to ensure this, library interfaces should be designed by more than one person, to avoid a tendency to bias the design to a single person's needs. This can be further enforced by having several test programs of similar functionality written by different programmers using the same libraries. If each programmer is able to use a given library interface in a clear, logical fashion, this is indicative of a well defined interface.

Chapter 4

Descriptions of Individual Library Modules

The Testbed libraries are organized as a three level structure. The lowest level consists of utility libraries. The middle level contains libraries of routines which perform operations on individual objects. The highest level libraries pertain to specific rendering techniques. Individual library modules which comprise each of these levels are described below.

4.1 Utility Level Modules

4.1.1 Transformations (XFM)

The XFM library consists of a set of tools for performing various vector and matrix operations. This is a very widely used library, called from almost every library on all three levels, and by most applications.

The main data structures defined are matrices (3 by 3 and 4 by 4) and vectors

(3D and 4D). Both are implemented as arrays of double precision floating point numbers.

Among the operations provided for vectors are component initialization, copying, addition, subtraction, multiplication and division by a constant, dot product and cross product, negation, linear combination, length, and normalization routines. It is possible to create any standard transformation matrix, such as identity, scale, translation, rotation about an arbitrary axis, or to create user initialized matrices. Standard matrix operations are furnished, such as matrix multiplication, premultiplication and postmultiplication of vectors with matrices, determinants, and inverse and transpose matrices. Most operations are implemented as macros, to allow maximum code efficiency.

4.1.2 Modeler Independent Description (MID)

This library reads MID format files, as described in Chapter 3, creating a standard run time structure needed by rendering applications. This process may include the reading of geometry attributes (through FED, etc.) and/or rendering attributes (through ATTR) associated with the given environment. Figure 4.1 shows how MID uses other libraries to read all data in files pertinent to a given MID file.

The main data structure used in MID is the *mid environment*, which contains all data pertinent to the environment. This includes the number of objects in the environment, a list of all these objects, and a list of attributes. A single pointer is maintained to this structure, allowing all data for a given environment to be easily accessed from this point.

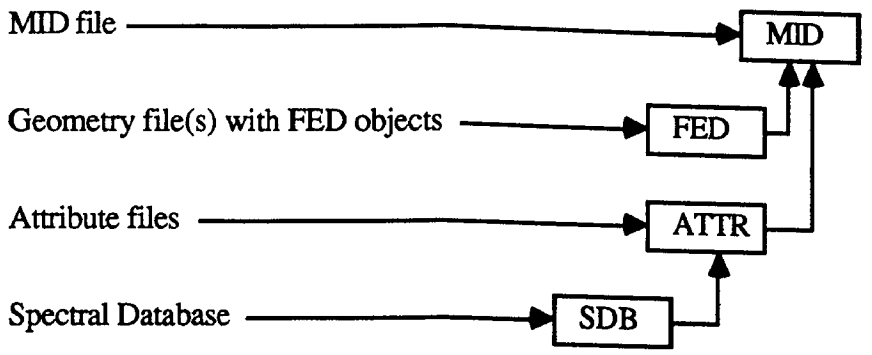

Figure 4.1: MID's uses of other libraries for reading environment data

There are several methods of reading a MID file. The simplest method just reads in the whole environment in a single call. This is sufficient for most applications involving small to medium sized environments. For very large environments, a sequential read mechanism can be used. Objects are read individually, allowing the renderer to process only certain parts of the environment, while not having the whole environment in memory at any given time. It is also possible to read in a specified subset of the environment. This can be done sequentially or in a single operation. Functions are provided to read files containing geometry attributes and rendering attributes, and to access macros which allow any of this data to be queried. Finally, a function is provided to delete environment structures from memory.

4.1.3 Attributes (ATTR)

The ATTR library allows the retrieval of rendering attributes which have been assigned to MID objects. Structures are created which contain references to SDB (Spectral Database) structures. These ATTR structures are passed by the renderer

to the SHAD library, or some other shading mechanism, which extracts the data needed by the particular lighting model being used.

An attribute is comprised of a *name*, a *material*, an *RGB* triplet, a complete *surface* description, and *emission* properties on a wavelength basis (See Figure 4.2). For simplification purposes, the parameters for a Phong reflection model are also stored. It is not necessary for each field to contain a value for a given attribute. A *material* contains color spectra for diffuse reflectance, specular reflectance, diffuse transmittance, and specular transmittance. In cases where simple lighting models which do not need these fields are used, the *RGB* triplet alone is sufficient. A *surface* contains data for diffuse reflectance, specular reflectance, diffuse transmittance, specular transmittance, and emission. For simple lighting models, the *Phong* structure is sufficient, and the *surface* properties are not used. The *emission* of a surface is described by its type (point, line, area), energy, and directionality information. The *emission* structure is included only if the surface emits light.

The attributes stored in a file are read into an an attribute array, and then accessed by name. These names are used in the MID file to refer to attributes. Once an attribute has been located, access macros can be used to retrieve values of any given field.

4.1.4 Spectral Database (SDB)

This library provides an interface to the Spectral Database, which stores data such as chromaticity spectra and other material attributes needed by rendering applications. The ATTR library makes calls to SDB routines when filling in attribute

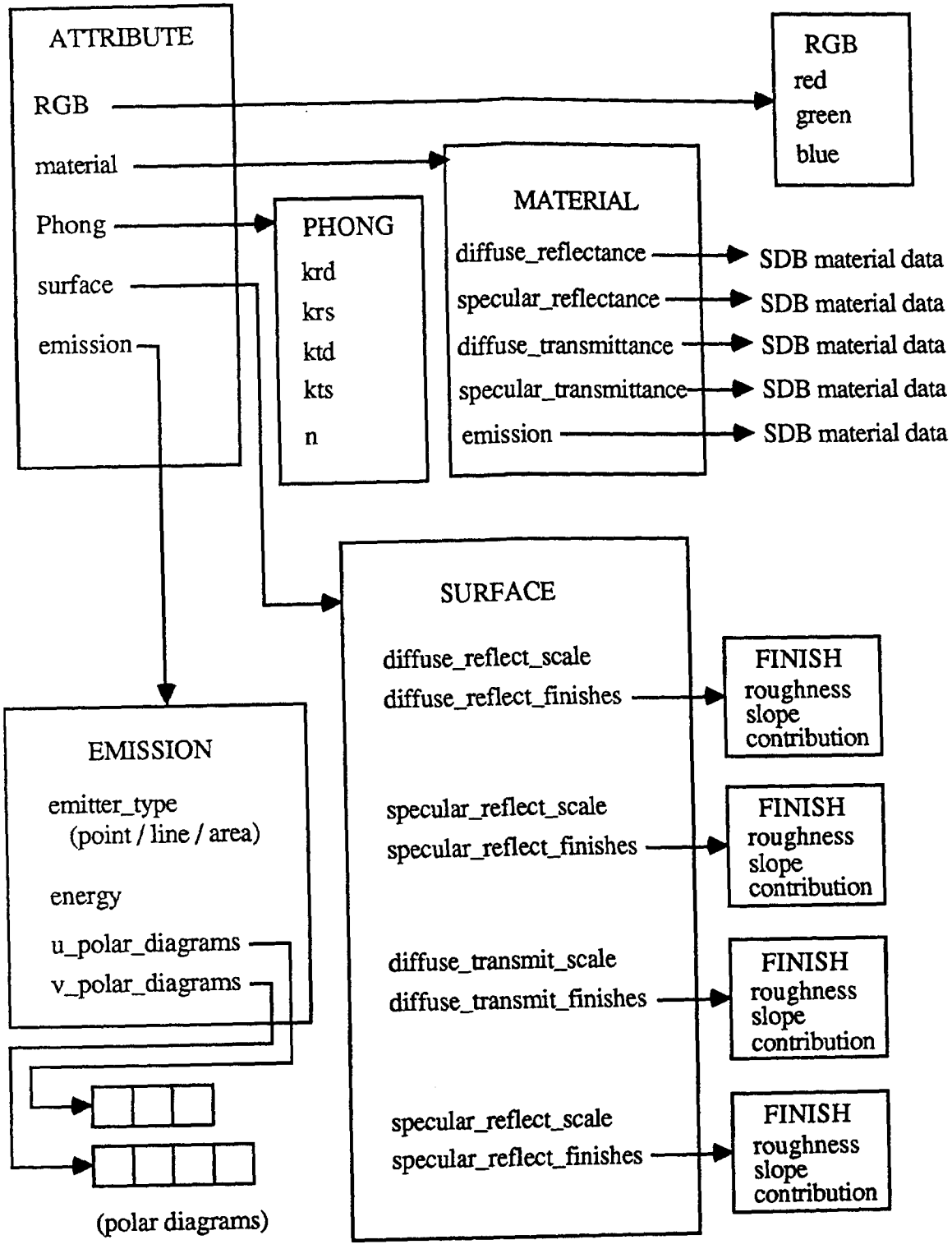

Figure 4.2: ATTR structures

structures.

The Spectral Database is a collection of files containing materials (Figure 4.3). Each material belongs to a class (such as building materials, naturally occurring

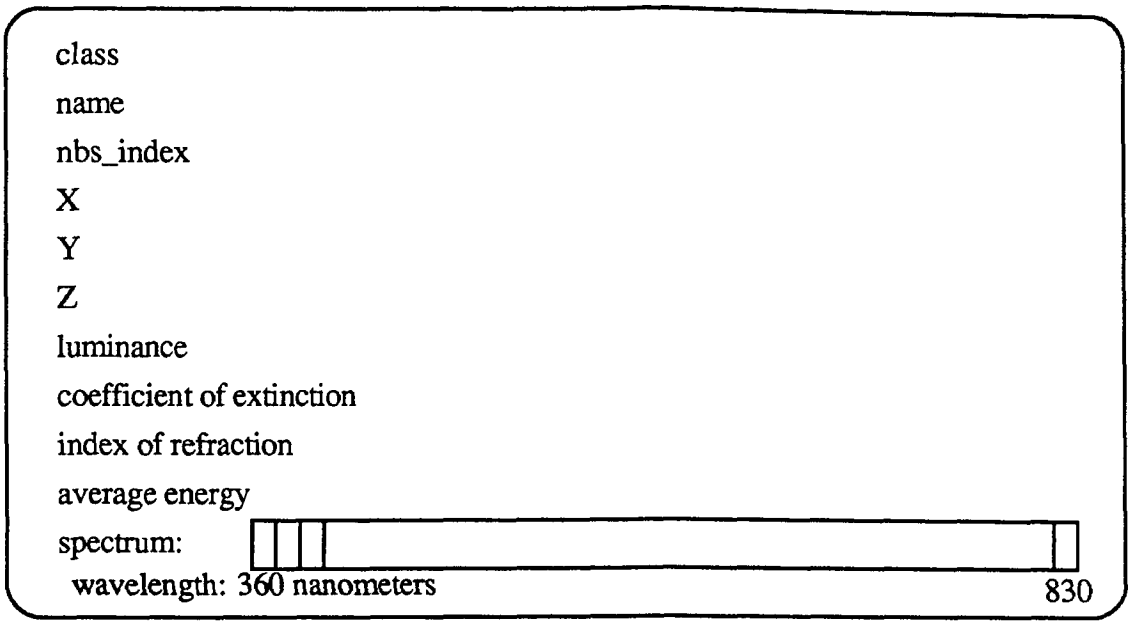

Figure 4.3: SDB material data

materials, or paint), and has a text name. The chromaticity spectrum is stored as samples at one nanometer increments from 360 to 830 nanometers. Each material also contains the National Bureau of Standards (NBS) index, the XYZ values for CIE color space, and the average energy for the spectrum. Some materials also store coefficients of extinctions and/or indices of refraction.

Routines are provided to access any or all material fields. In addition, a list of all classes or all names in a given class can be compiled.

The routines contained in the SDB library where originally written by Roy

Hall in 1983, based on work done by Gary Meyers. Recently, the file structure and interface were completely redesigned. However, all the original spectral data was retained.

4.1.5 Color Space Operations (CLR)

The CLR library provides tools used for color space transformations. It is used by renderers to perform device independent color calculations in a choice of different color spaces, and then to transform to RGB values which are proper for a given monitor.

The CLR library is based on the CIE XYZ system. Colors represented in all supported color systems can be converted to and from this system, with the exception of the hexcone system, which converts to and from the linear luminance RGB. The supported color spaces include the Munsell system, the CIE LAB system, the CIE LUV system, HCV, and NTSC's YIQ system.

Figure 4.4 shows the flow of transformations provided by this library. Spectral energy distributions can be converted to XYZ tristimulus values. The XYZ values can then be converted to chromaticity and luminance, or any of the supported color spaces. These can also be converted back to XYZ color space coordinates. The XYZ values can be transformed into linear luminance RGB values, which can in turn be transformed into the RGB values necessary to provide the linear voltages given the characteristics of a specific monitor.

This library is based on work by Gary Meyer [MEYE83] [MEYE86].

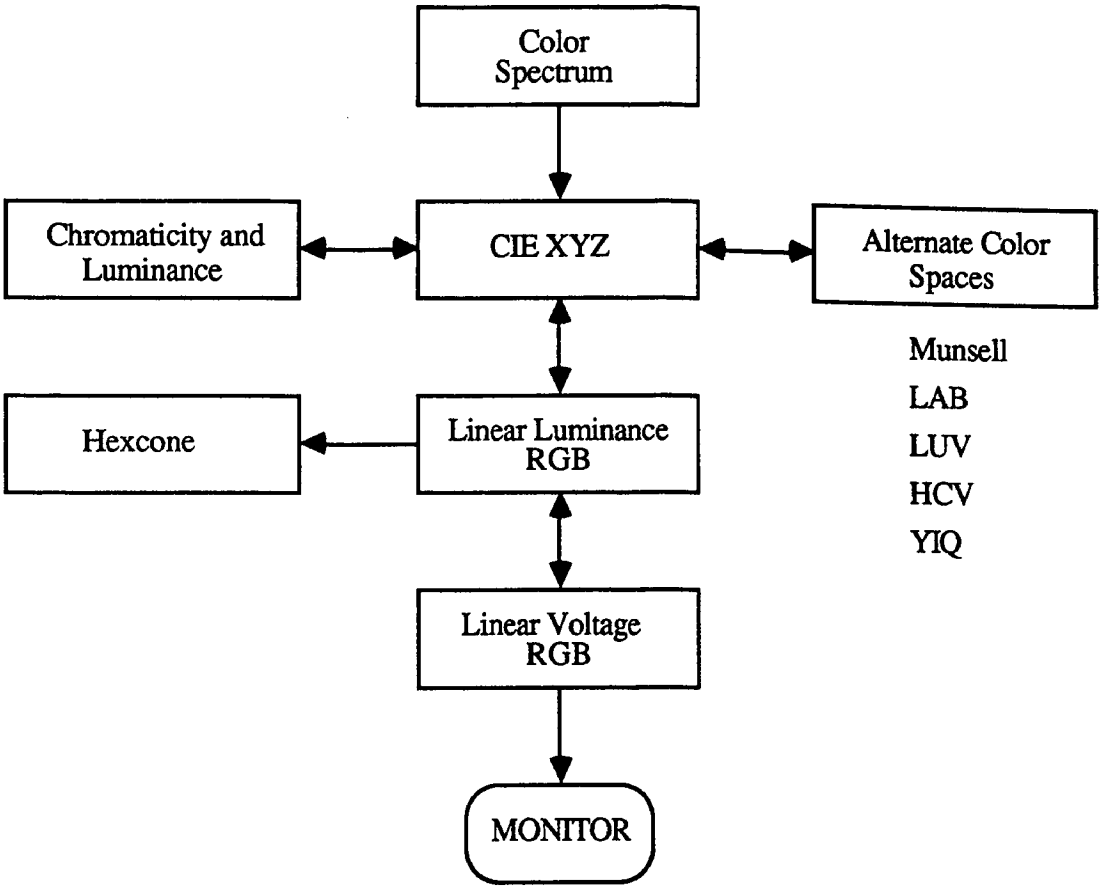

Figure 4.4: CLR color space transformations

4.1.6 Ray Operations (RAY)

The RAY library contains definitions of the data structures for both a ray and a view volume which contains virtual screen resolution data. Routines are provided to initialize the viewing volume structure (based on a transformation matrix for a canonical camera with the eye position at the origin), and to initialize first generation rays (eye rays) and child rays (reflected and refracted rays). The RAY library is used in any application involving ray tracing, and is used by both Level

2 libraries (RIN) and Level 3 libraries (SRT, HBV, USS, ANT, MC).

A *ray* is comprised of an *origin*, a *direction*, a unique *ID*, the *depth* in the ray-tree, and the current *total attenuation*. The *view volume* structure holds the view *transformation matrix*, the *eye position*, image plane resolution data, *forward direction*, *maximum ray-tree depth*, and a *minimum attenuation cutoff* (Figure 4.5).

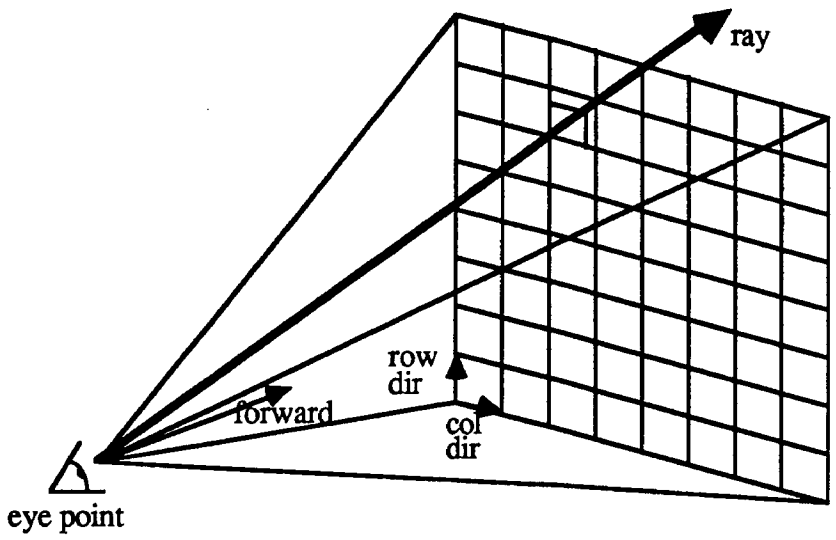

Figure 4.5: RAY library's view volume

The initialization routine fills the *view* data structure given the *camera transformation* matrix, number of rows and columns, *maximum ray-tree depth*, and *minimum attenuation cutoff*. Several ways to initialize the *ray* structure are provided. A function is provided to initialize a first generation ray given the image plane pixel row and column, and optional offsets to allow shooting through arbitrary points on the pixel (Figure 4.6). There are also functions to spawn new reflected and refracted rays. These functions take as input the incident ray, the

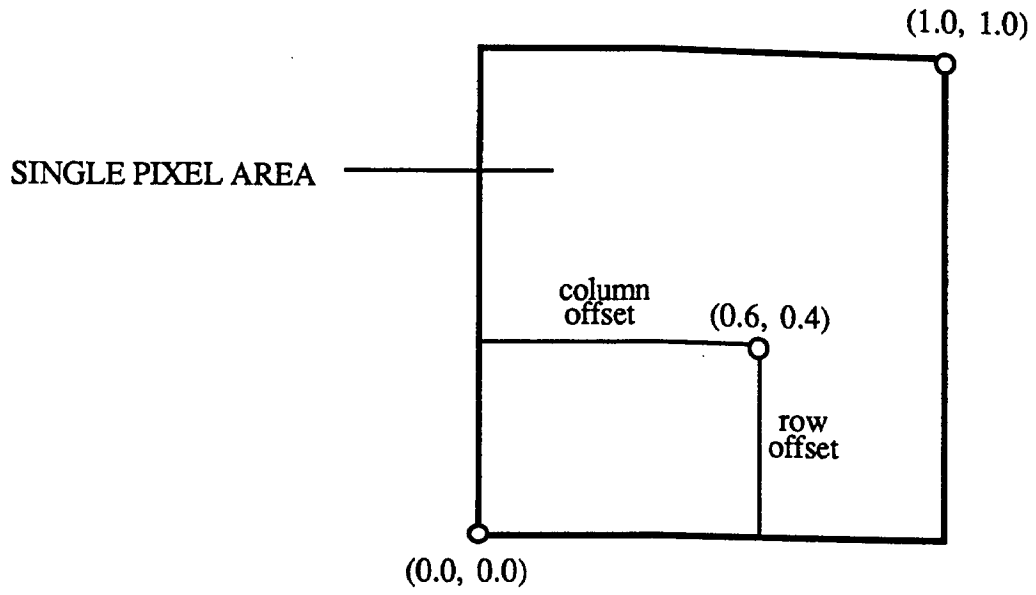

Figure 4.6: Row and column offsets in a pixel

surface point, the surface normal, and the surface properties, and will create the desired new ray (Figure 4.7). A function is provided to fill the structure with user supplied origin and direction vectors. There is also a function to return the world coordinates of the intersection point on an object's surface, given the ray and the distance traveled, which can be used to calculate the origin of the next generation ray.

4.1.7 Run Length Encoded Images (RLE)

The RLE library provides a standard Testbed interface to the Utah Raster Package, which is described in The Utah Raster Toolkit [PETE86] and Design of the Utah RLE Format [THOM86]. It allows the creation, reading, and manipulation of images stored in Run Length Encoded format, and is used in the output stage by

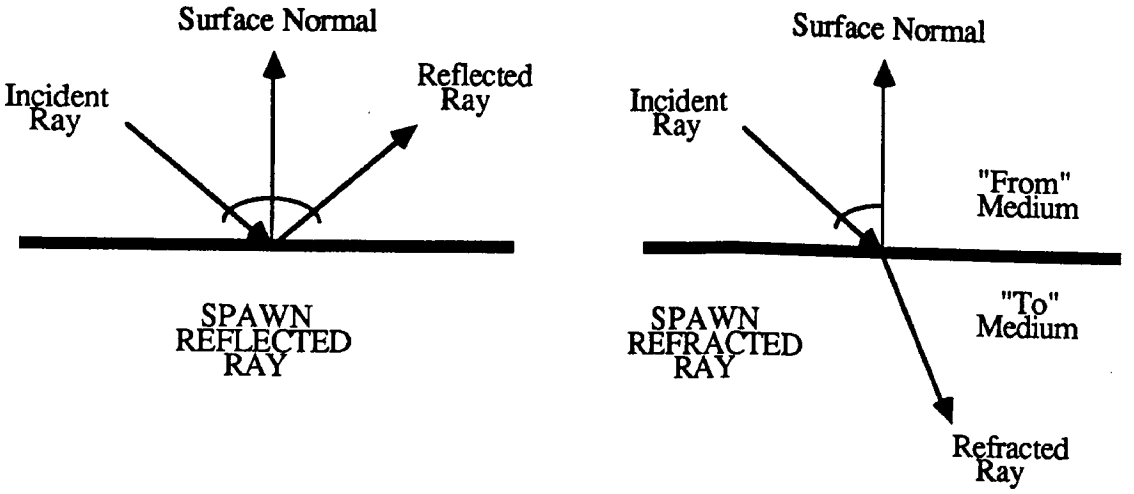

Figure 4.7: Spawning reflected and refracted rays

all renderers.

Pixels can have either 8-bit or 16-bit channels, and an optional alpha buffer channel. Scanlines consist of a user specified number of pixels.

For both pixels and scanlines, functions are provided for memory allocation, reading from files, and writing to files. Also, mechanisms are supplied to read and write information stored in the RLE file headers.

Several utility programs have been written as auxiliary parts to this library. These allow operations such as displaying a raster image at a screen location specified by using a tablet, querying the RGB values at a given location, and interactively enlarging a portion of an image to examine details.

4.1.8 Face-Edge Data Structure (FED)

The FED library contains routines used to create and manipulate the topological descriptions of objects defined by the Face-Edge Data structure [WEIL86]. This library handles only the topological description; the geometrical description is described by FGN structures. This object description format is used by all Testbed modules that manipulate polygonal data.

Figure 4.8 shows an example of a very simple model and the corresponding topological representation. Figure 4.9 depicts the data structures used to represent this example.

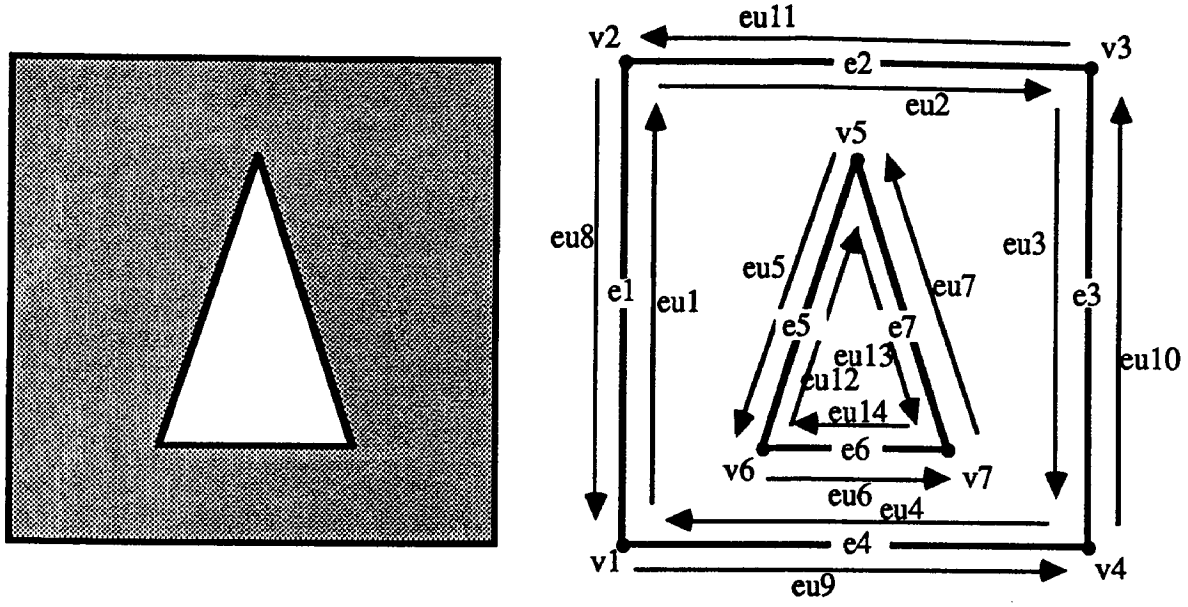

Figure 4.8: Example model: a square face with a triangular hole

The topology of an object is described by a hierarchical system. The highest level of the description is the *model*, which is comprised of one or more *shells*, which

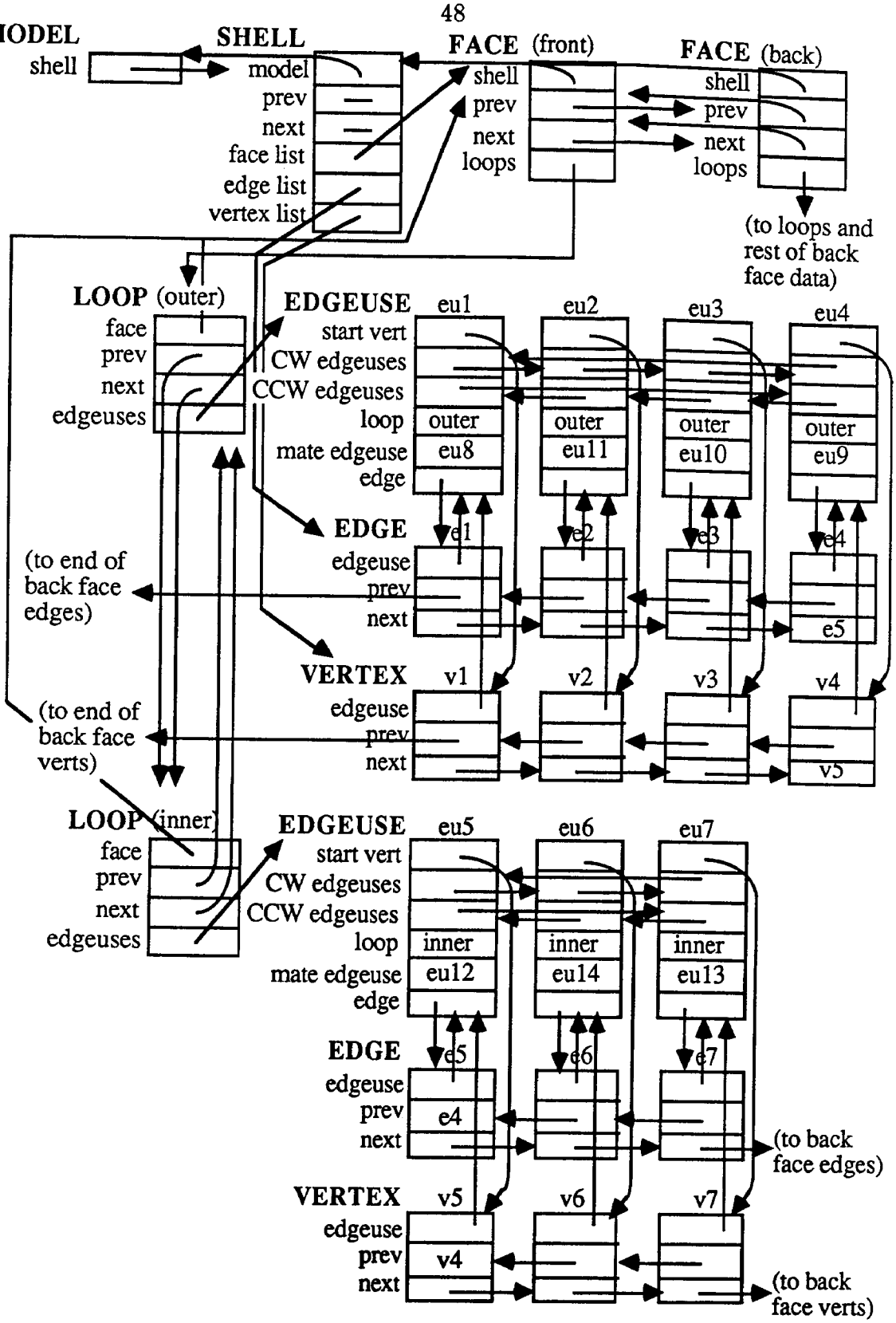

Figure 4.9: The FED data structures used to represent the example model

each in turn consist of a number of *faces*. Each face is described by one or more *loops*, which are each defined by *edgeuses*. An edgeuse is described by its two end *points*. If a face has more than one loop, it is assumed that the additional loops describe holes in the face. At all levels, bi-directional pointers are maintained to adjacent elements.

A series of construction and destruction operators are provided to build FED descriptions incrementally while always maintaining a consistent, valid topology. Shells, loops, faces, edges, and vertices can be created and destroyed. Functions exist to allow querying of the data structure for such information as the number of loops in a face, or the pointer to the previous or next loop in the face. Access routines allow the traversal of the structure on all levels. Read and write routines provide interfaces to binary files.

4.1.9 FED Geometry Nonplanar (FGN)

This library contains routines to allocate memory for the geometrical descriptions of objects defined by the Face-Edge Data structure. This is the counterpart to FED. All applications using FED descriptions must also use FGN in order to describe the geometry. The use of FED and FGN together allows the definition of a very wide range of objects. Most polygonal and non-planar objects can be defined with this method.

Each element of the FED data structure maintains a list of attributes. Geometry is simply one of these attributes, for which this library allocates and deallocates memory. Currently, faces and edges are limited to planar polygons and lines. When

nonplanar geometry is supported in the future, nonplanar faces and edges will be described using NURBs (see the NURB library description).

4.1.10 FED Attribute Manager (FAM)

The proposed FAM library will provide routines to create, access, and modify attributes stored in the FED topology description. Attributes may be assigned at any level in a FED object. These attributes include geometry attributes, handled through FGN and rendering attributes defined by ATTR, but may also be user-defined application specific attributes. The routines in this library will maintain attribute lists at each level, and provide an interface for individual attribute types.

4.1.11 Non-Uniform Rational B-Splines and Patches (NURB)

This proposed library will provide tools used for creating and manipulating Non Uniform Rational B-Splines, and will initially be used to define nonplanar FED objects.

One function will take as input a set of control points and create parametric functions for curves and surfaces. Also, functions will exist to calculate the XYZ values (object coordinates) given UV parametric values, and to calculate UV parameters given the distance along a curve, T.

4.1.12 Newton's Method (NEWT)

The NEWT library contains routines to find roots of univariate polynomial equations, using Newton's method. This is useful for some ray intersection calculations

such as those for tori and spline surfaces.

An array of coefficients is passed to the root finding functions, which returns the number of rational roots found, filling in an array with these values. A maximum number of iterations can be specified, as can the *epsilon* error term. The root finding functions are guaranteed to terminate.

4.1.13 Gauss-Siedel Solver (GS)

This library solves systems of linear equations using the Gauss-Siedel iterative solution method. It is useful in solving those linear equations involving radiosity form factors.

A matrix of coefficients, an initial guess vector, a size, and a relaxation parameter are passed to the solver, which returns a solution vector.

4.1.14 Camera (CAM)

The proposed CAM library will provide a general way to define and manipulate cameras. Parameters will include the viewing frustum, hither and yon planes, various lenses, apertures, and focus. CAM will contain routines used to convert viewing transformations to camera parameters, and visa versa.

4.2 Object Level Modules

4.2.1 Ray Intersection and Normal (RIN)

The RIN library contains the definition of the data structure for RIN objects. These structures are used by the RIN functions which calculate ray-object intersections and surface normals.

Each RIN data structure contains the object type, an object-space to world-space transformation matrix and its inverse, and various miscellaneous fields such as the last intersection point, distance traversed, and the ray ID. Also, any object specific data is stored here, such as plane equations, precomputed surface normals for planes, vertices, radii, and other parameters. In addition to the above data, each RIN data structure stores a pointer to its intersection and normal calculation routines.

The three basic routines provided in this library are initialization, intersection calculation, and normal calculation, and are outlined in Figure 4.10. The initial-

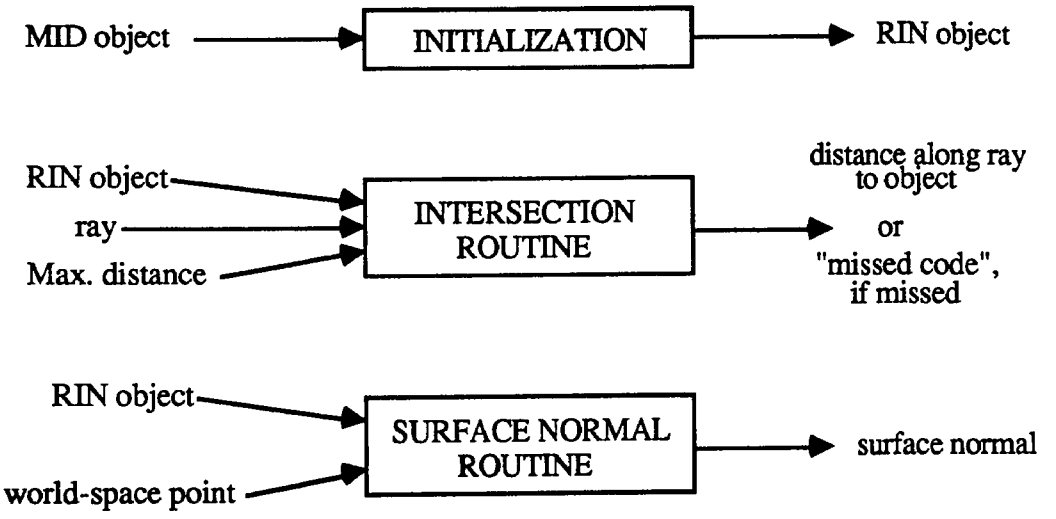

Figure 4.10: RIN routines

ization routine creates a new RIN object data structure given a MID object, and fills in any data that can be precalculated. The intersection routine determines the closest hit for a given object and ray. The normal calculation routine finds the surface normal at a given point in world space.

4.2.2 Bounding Volumes (BV)

This library creates Bounding Volumes for all supported types of MID objects, and performs ray intersection operations with them. The two types of bounding volumes currently supported are spheres and rectangular boxes (which are aligned with the axes of the environment). For MID objects, the bounding volumes are created so as to be as tight fitting to the object as possible. The creation routines will supply either spherical, rectangular, or "best", which is defined as the one with least volume.

The bounding volume data structure contains data for either spherical or rectangular geometric data, as well as pointers to the appropriate intersection functions. This allows the applications programmer to treat the bounding volume generically after its initial creation, without being concerned with its type.

For both kinds of bounding volumes, two intersection routines are provided (see Figure 4.11). The first returns true if the given ray intersects the bounding volume is less than a supplied maximum distance, while the second supplies the distance to the intersection if the ray does in fact intersect the bounding volume.

4.2.3 MID to FED Conversion (MFC)

The MFC library performs MID object to FED object conversion. This is useful for applications which operate exclusively on FED objects. An example of this involves the radiosity routines, which require FED representations of all objects (see Figure rad).

A MID object of any supported type is input into the converter. The output is a

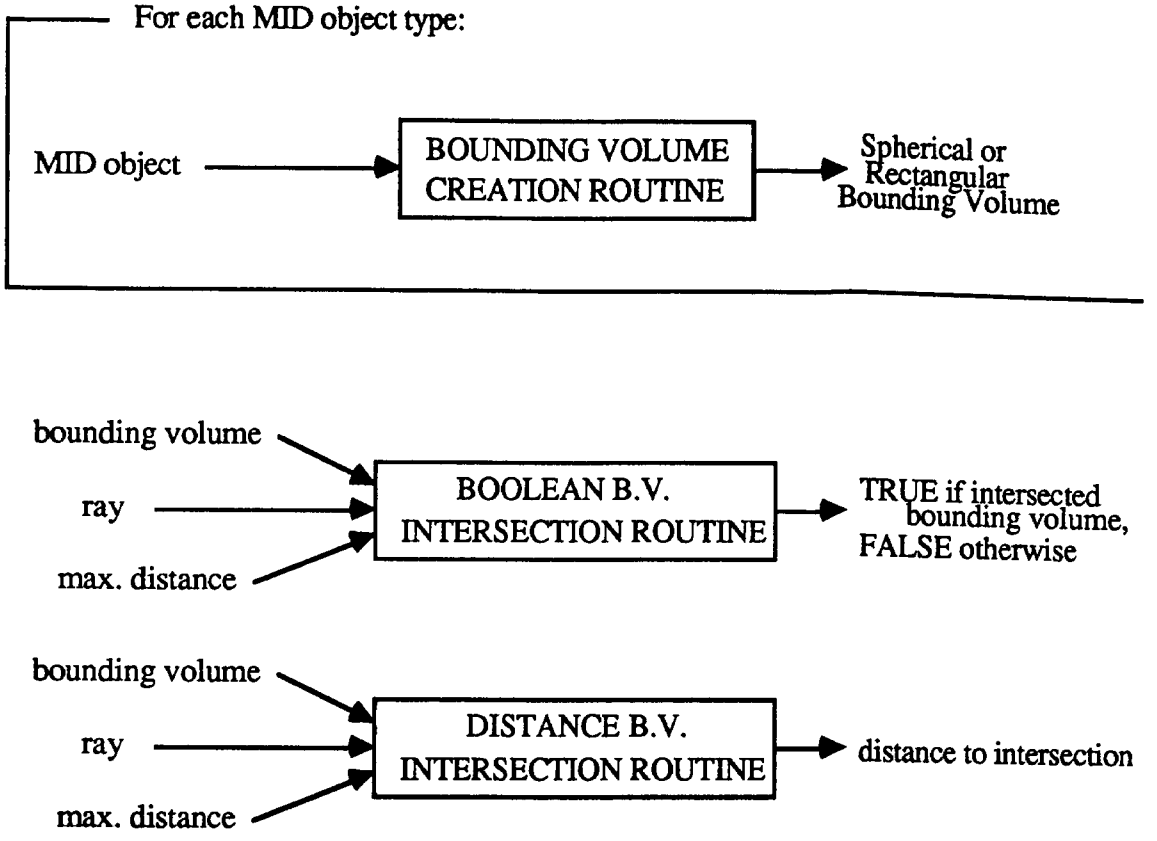

Figure 4.11: BV routines

new FED object which exactly represents the input object, even if the input object is comprised of some curved surfaces. In this case, the nonplanar features of FGN would be used to maintain an exact representation. Only the planar version of this library has been implemented, and is called PAP, for Polygonal Approximation.

4.3 Rendering Level Modules

4.3.1 Simple Ray Tracing (SRT), Hierarchical Bounding Volumes (HBV), Uniform Space Subdivision (USS)

These libraries each supply the same functionality, but differ in their implementation and efficiency. All three libraries provide high level ray tracing routines which determine the closest object hit by a ray, and the shadow status of a given surface point for a given light.

- SRT (Simple Ray Tracing) stores the RIN objects in a single level array, providing the functionality of the most naive ray tracer.
- HBV (Hierarchical Bounding Volumes) places bounding boxes aligned with the axes around objects and clusters of objects in a hierarchical fashion.
- USS (Uniform Space Subdivision) subdivides space into uniformly sized rectangular regions, each holding a list of objects which are at least partially contained by the region.

The second and third libraries offer substantial computational savings for ray tracing at the expense of additional storage. The applications programmer can select which library to use based on the particular needs of the application. Because their functionality and interfaces are identical, these libraries are completely interchangeable providing maximum flexibility.

The data structures are quite different for each library, but are transparent to the applications programmer.

The interface to these libraries consists of three routines (see Figure 4.12). The

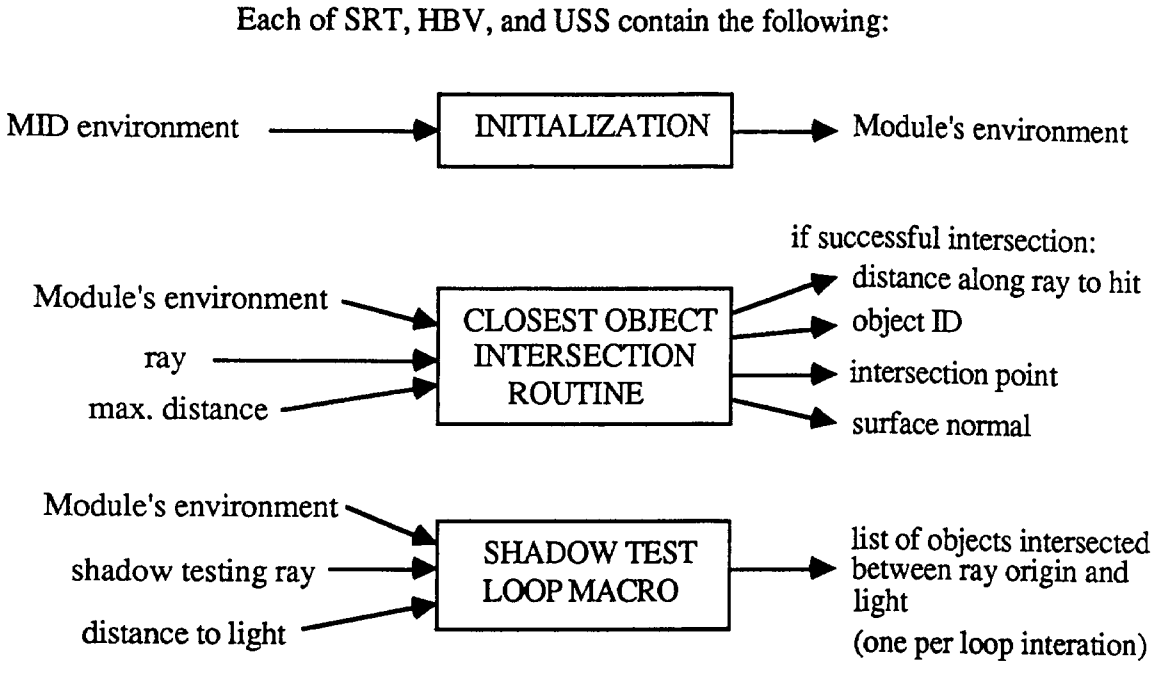

Figure 4.12: High-level ray tracing libraries (SRT, HBV, and USS)

initialization routine takes a MID environment and returns a pointer to a special structure initialized by the given library. The intersection routine takes as input a ray and an initialized environment, returning the distance along the ray to the closest object hit, the object's ID, and the surface point and normal in world space coordinates. A macro is provided which loops through each object hit between the ray origin and a given light position. This routine is useful in shadow testing where the application can check for opaque objects and exit the loop immediately if one is found.

4.3.2 Antialiasing (ANT), Monte Carlo (MC)

These libraries contain tools useful for performing sampling of continuous functions in a ray tracing context. This may be used to sample image plane information such as pixel data, or object space information such as reflection and refraction integrals.

The ANT library provides two methods of antialiasing: adaptive pixel subdivision, and stochastic sampling [COOK86]. These methods are implemented as high level control macros (see Figure 4.13). The origin and direction of top level

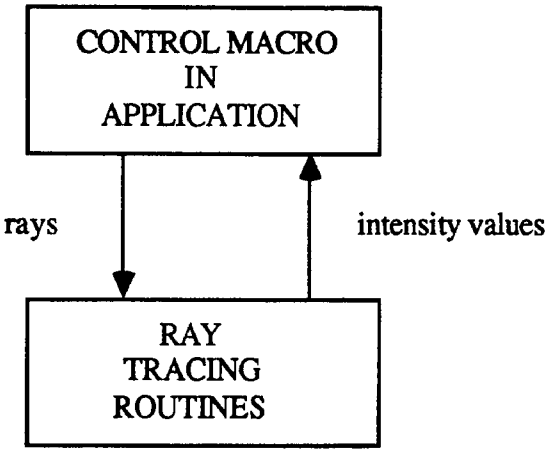

Figure 4.13: ANT library's antialiasing loop macro

rays are set by the macro and then passed to the application, which returns an intensity. The functionality provided essentially consists of an evaluation of the current intensity results, and a decision making scheme which decides where to shoot the next ray if more sampling is needed. The macros continue to pass rays to the applications until enough samples have been taken to satisfactorily represent the intensity at that particular pixel or point on a surface.

The two methods differ only in the technique used for choosing new rays. In

adaptive subdivision, pixels are recursively subdivided into subpixels and rays are shot through pixel corners (Figure 4.14a). This process continues until the values at adjacent sample points are within a specified threshold. In stochastic supersampling, new rays are shot according to a random distribution function covering the pixel area (Figure 4.14b).

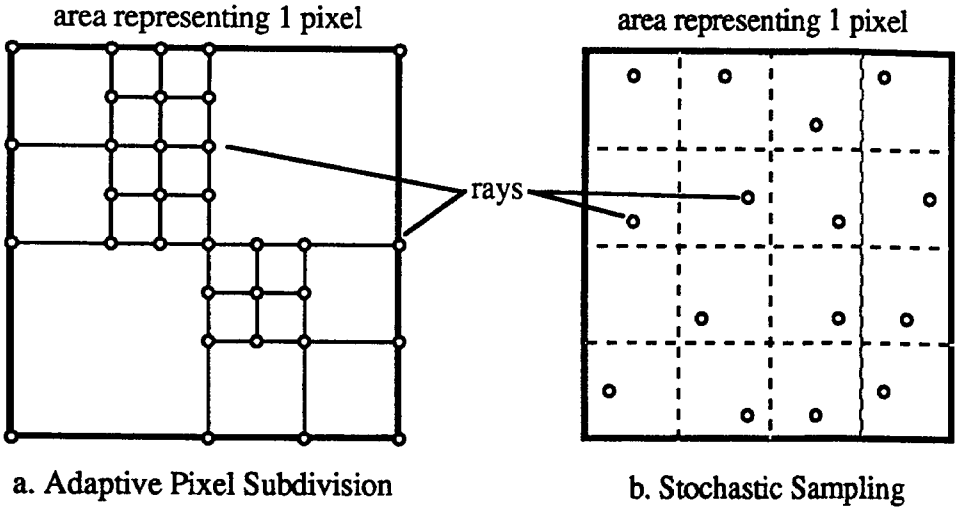

Figure 4.14: ANT library's pixel supersampling schemes

The proposed MC library will provide routines to evaluate reflection and refraction integrals, which function in a way similar to those in the ANT library. Additional information beyond intensity may be needed in order to make the decision about what ray to shoot next. This environment specific information will be supplied by the application.

4.3.3 Shading (SHAD)

The SHAD library provides routines for determining the shading at a surface point. Several lighting models will eventually be supported, including the Phong, Hall, and Cook-Torrance models. Routines in this library are used by ray tracing applications to calculate the color intensity at a given node in the shade tree.

A typical lighting model will take as input the surface normal, view direction, various surface attributes, the direction to the light source, and its emission properties. A color is returned, represented in RGB, XYZ, or some other color space (see Figure 4.15).

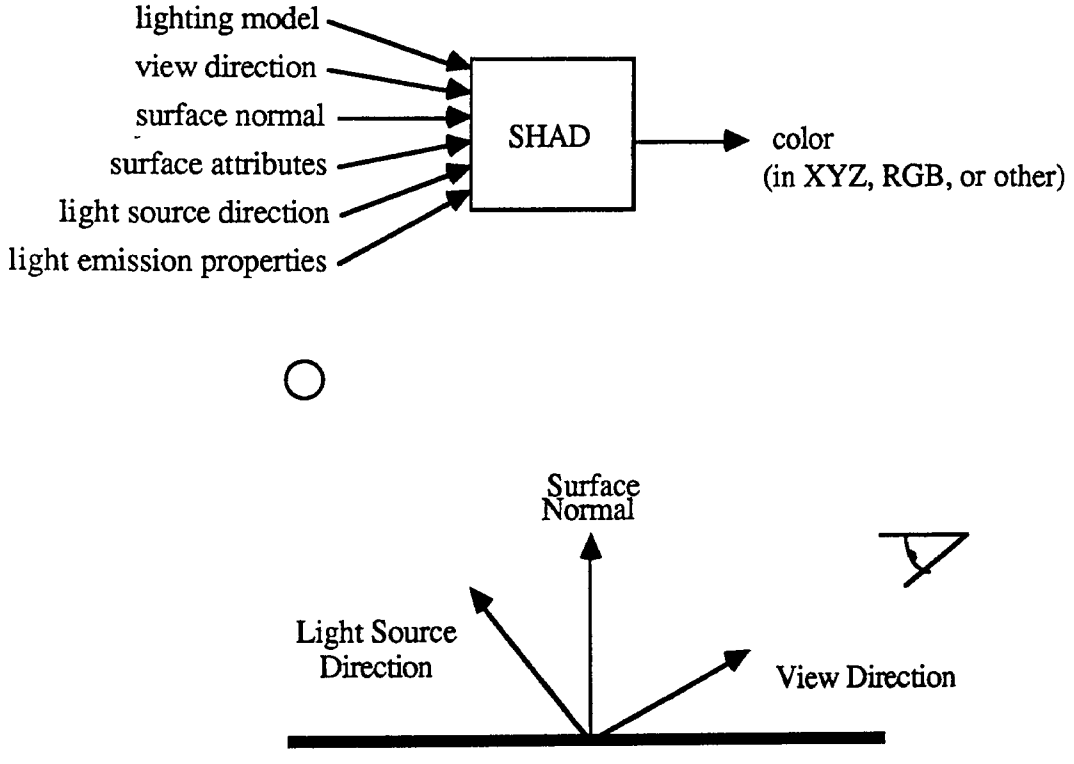

Figure 4.15: SHAD library functionality

4.3.4 Texture Mapping (TXM), Texture Sampling (TSM)

These libraries provide tools that perform various texture functions. It is possible to use these libraries to map textures of color, bumps, reflectivity, transparency, and other properties onto a given surface.

The TXM library contains routines which, given a RIN object and an intersection point, calculate the normalized UV parameters for that point on the surface. These UV parameters are passed to TXS routines and used as coordinates of a sample point on a texture map. In the future, the functionality will be extended to handle mapping various sections of bitmaps onto arbitrary surfaces, and to create procedural textures.

4.3.5 Radiosity Algorithms (RAD)

This proposed library will be the highest level module in the radiosity system (depicted in Figure 4.16), and will provide an interface to the radiosity solution algorithms. It will contain a few routines which calculate a full radiosity solution, and many others which could be used collectively as a "toolkit" for the development of future radiosity algorithms. RAD will call routines from the ASA library to perform adaptive mesh subdivision using MQT, and routines from FFC to execute form-factor calculations. The GS library will be used to obtain a solution for the gather method.

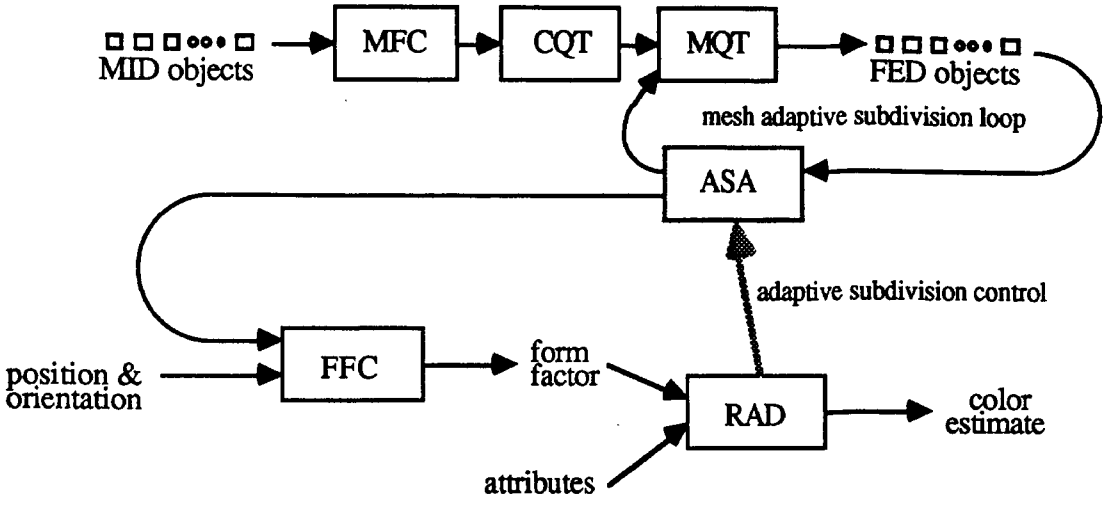

Figure 4.16: The function of RAD within the context of the radiosity libraries

4.3.6 Adaptive Subdivision Algorithms (ASA)

The proposed ASA library will provide the necessary tools to adaptively subdivide an environment of FED objects. A quad-tree data structure will be defined where nodes in the tree will point to four child nodes which represent either quadrilaterals or triangles. Leaves of the tree each point to a FED face which is either a quadrilateral or triangle.

As a pre-process, the library will use MFC to convert primitive MID objects to FED, and CQT (the mesh creation library) to create the initial mesh of quadrilaterals and triangles. Then, MQT (the meshing library) will be used to dynamically subdivide the leaf or node into four quadrilaterals or triangles. The ASA library will also contain "decision-making" routines used to decide whether the sub-surface of a face should be further subdivided.

4.3.7 Create Quads and Tris (CQT), Mesh Quads and Tris (MQT)

These proposed libraries perform surface meshing, and are mainly intended to be used in radiosity applications. The CQT library will be used to create the original mesh of quadrilaterals and triangles, while the MQT library will perform further meshing on those elements.

CQT is given an arbitrary FED model (which can be concave or convex, and may contain holes), and returns a FED model which is meshed into quadrilaterals and triangles. The mesh routine attempts to generate quadrilaterals whenever possible, which helps to improve the quality of the radiosity solution. MQT takes a FED face which is a quadrilateral or triangle and subdivides it into four sub-faces of the same type (See Figure 4.17). This operation is performed by creating only

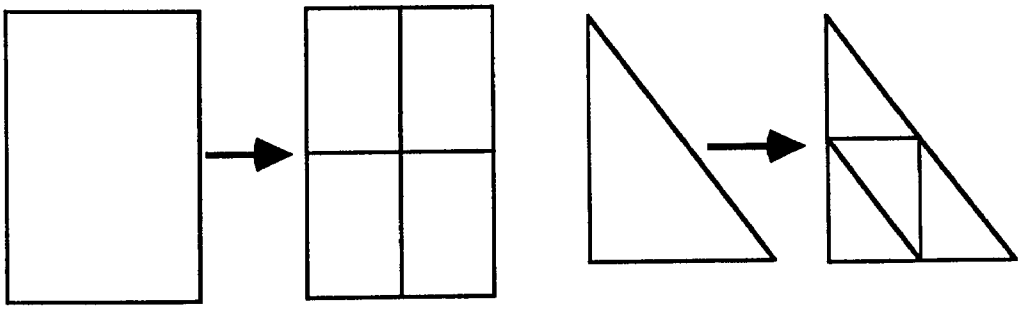

Figure 4.17: Quadrilateral and triangle meshing

three new faces and modifying the original face to account for the fourth sub-face.

4.3.8 Form-Factor Calculations (FFC)

The proposed FFC library will perform numerical techniques, such as the hemi-cube operation, to compute form factors used for the radiosity solution. These routines will take as input a point on a surface, the surface normal, and list of FED faces representing radiosity elements or patches. The output will be a vector of form factors.

Among the techniques to determine form factors will be the hemi-cube operation which scan-converts the objects in the environment onto each face of the hemi-cube using calls to IBUF.

4.3.9 Item Buffer (IBUF)

The proposed IBUF library will contain routines to construct and access data in item buffers (see Figure 4.18). An item buffer is very similar to a Z-buffer. For an arbitrarily sized image plane, each pixel contains an entry. Instead of an intensity, as in the case with a Z-buffer, an item buffer entry holds the ID of the closest visible surface. Item Buffers are useful for hemi-cube calculations in radiosity, as well as in the pre-process stages of ray tracing [WEGH84], as described in Chapter 2.

The functionality of this library will include allocation and initialization of any number of item buffers, allowing applications to work with an arbitrary number of them concurrently. The interface will allow changing item buffer attributes, and accessing and changing the item buffer and Z-buffer memories. The types of objects supported by the scanning routines will include triangles, convex quadrilaterals, general polygons defined by a list of vertices and a normal, any MID object, a FED

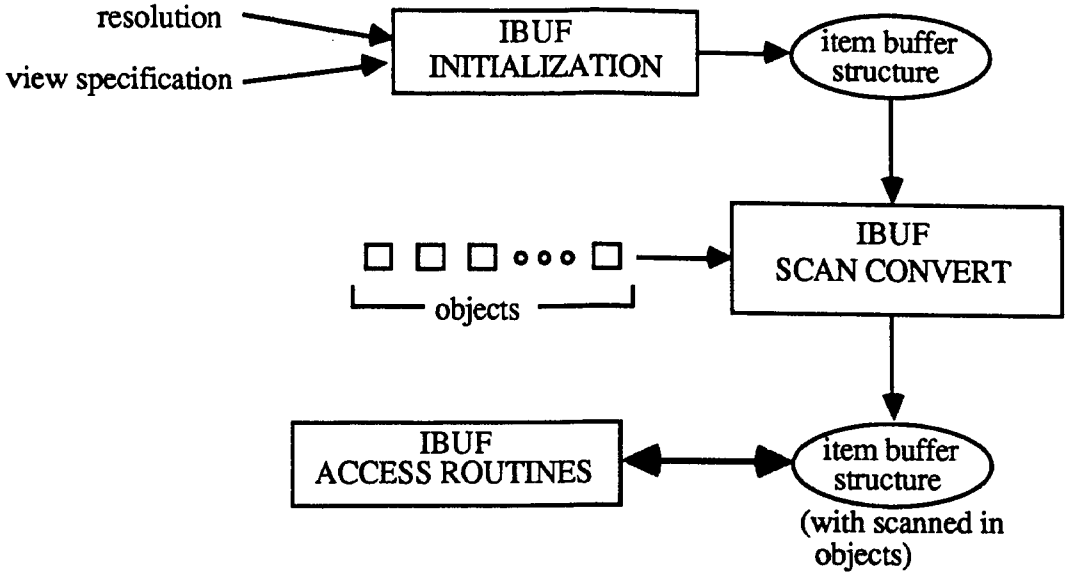

Figure 4.18: Functionality of IBUF library

surface defined by a list of FED vertices and a normal, and a FED face. Routines will also be provided which allow the application to compose two item buffers in a variety of ways.

Chapter 5

Example Application Using Testbed Libraries

This chapter describes a ray tracing application implemented by the author, which makes extensive use of libraries from the Testbed. More than 90 percent of the code comprising this program is actually contained by Testbed library modules, while the remainder controls various options in functionality, and some user specific features. This application was written as a test program and was used during the development of the Testbed libraries. It served as a debugging tool, as well as an example of the use of the Testbed.

5.1 General Description of Application

The application implemented is a ray tracer [WHIT80,ROTH82]. Ray tracing is a process which involves tracing the paths of light rays from a given eye point through pixels on a view plane, and back though an environment. Rays are reflected from

and refracted through surfaces. An intensity contribution at each ray-surface intersection point is calculated based on the position and emission properties of the light sources in relation to the point.

In this ray tracer, all object types described by MID are supported, including spheres, cones, cylinders, cubes, conics, torii, prisms, pyramids, and FED objects. The ray tracer performs reflection and refraction to a user specified ray tree depth or maximum intensity attenuation level. The Phong lighting model is supported, and is enhanced by allowing filtering of the light spectrum by transparent objects. Simple surface texture mapping routines for most object types are supported. Two kinds of antialising are supported in this implementation: adaptive pixel subdivision and stochastic sampling [COOK86].

Bounding volumes are used for all objects, and two acceleration methods are provided. Either hierarchical bounding volumes [WEGH84,GOLD87] or uniform space subdivision [FUJI85,CLEA87] may be invoked, or a naive method involving no acceleration may be used. The latter method is provided as a control to compare the performance of different acceleration techniques.

This program outputs an image in run length encoded format (RLE). The input to the program is a MID file describing the environment to be rendered. This MID file may be generated by one of several existing modelers which support MID, a text editor, or a procedural environment generator which outputs a MID file. Any attribute files, image files (for use in texture mapping), or object geometry files referenced in the MID file will be read by the program. Also, a setup file is read, which supplies a number of parameters required by the ray tracer.

The parameters in the setup file include the output image resolution, as well as the starting and ending scanline and starting and ending pixel, used when rendering subsections of the image. The maximum ray tree depth, and maximum attenuation cutoff are specified, as well as the antialiasing parameters. The chosen acceleration method and lighting model are also selected here.

The routines in this ray tracer were implemented in such a way as to provide a simple interface to initialization and pixel color calculation. This allows the construction of a control program which distributes the ray tracing calculations over a network of processors [TAMP89].

5.2 Program Functionality and Library Usage

The mainline program consists mostly of calls to routines in Testbed libraries. A simple pseudo-code version of the ray tracing process is depicted in Figure 5.1. First, the initialization is performed, which involves loading all data pertinent to the environment being rendered, and initializing various structures. When this is complete, the pixel intensities for each scanline are calculated and written out to file. The intensity of each pixel is determined by taking one or more samples.

The intensity of a sample for a given point on the viewing plane is determined as follows. A ray is shot from the eye point through the given point on the viewing plane pixel. Intersection tests are performed with objects in the environment, and the closest object hit by the ray is determined. If no object was intersected, then a background color is assigned to the pixel. Otherwise, the surface normal at the intersection point is determined and is used to perform shading calculations.

```
1 Initialization. {MID,FED,FGN,ATTR,RIN,BV,TXS,ANT,RLE}

2 Open image file. {RLE}

3 For each SCANLINE: {ANT}

4    For each PIXEL: {ANT}

5       For each SAMPLE: {ANT}

6          Shoot ray from eye point through view frustum. {RAY}

7          Determine closest object intersection. {SRT|HBV|USS,RIN}

8          If there was an intersection,

9             Find surface normal, {RIN}
10             do shadow testing, {SRT|HBV|USS}
11             perform texturing and surface shading. {TXS,TXM,ATTR,SHAD}

12            If reflective surface intersected, {ATTR}
13               shoot ray in reflected direction, {RAY,ATTR}
14               and continue recursively from line 5.

15            If refractive surface intersected, {ATTR}
16               shoot ray in refracted direction, {RAY,ATTR}
17               and continue recursively from line 5.

18         Else
19            Assign background color. {ATTR}

20   Write out full scanline. {RLE}

21 Close image file. {RLE}
```

Figure 5.1: Pseudo-code version of ray tracer

This process involves shooting one ray at each light source (assuming they are point sources) and again performing intersection tests with the objects in the environment. If an opaque object occludes the light, then there is no contribution for that light source. The color of the object at the given intersection point is also dependent on the texture map sample taken for that point on the object's surface.

If the object struck is either reflective or transparent, the respective contributions of each are now added by recursively repeating the above process of finding the closest intersection point and performing shading calculations. This process is terminated either by the current ray missing all the objects, reaching the maximum ray tree depth, or by surpassing the maximum intensity attenuation cutoff [HALL83].

The implementation of this process involves calls to routines in many different Testbed libraries. Subsections of the process and their usage of library routines is described in the following sections.

5.2.1 Initialization

The initialization stage, line 1 in Figure 5.1, begins by reading the setup file and filling the setup structure with the appropriate parameters. All environment data is then loaded. First, the MID file itself is read by a MID routine, and the MID environment data structure is built which contains all object data and all references to geometric data, and attribute data. A second MID routine is called to read each file containing geometric data. This routine makes calls to FED and FGN in order to read in FED object data. Another MID routine is called to read rendering

attribute files. This routine fills the ATTR structures with the appropriate data.

At this point the application extracts the camera view transformation from the MID structure, and initializes the view volume structure in the RAY library, accessed when firing eye rays. The ambient light color and all light source locations and colors are extracted from the MID structure. The user definable attribute feature of MID is used to associate an image file name with each textured surface. This information is extracted from the MID structure. Image files for texture mapping are loaded by TXS which performs various texture sampling initializations.

Next, the environment is initialized for ray intersection calculations. One of the three environment intersection libraries are used (SRT, HBV, or USS; see Chapter 4). The enironment intersection initialization involves several steps. First, for each MID object in the MID environment structure, a RIN object is created, which contains parameters used in the ray intersection and normal calculations. This includes precalculated normals for flat surfaces, and other values used in the ray intersection routines. Also, a bounding volume for each object is constructed by BV. These RIN objects and bounding volumes are then assembled into a structure specific to the acceleration technique used. This preprocess stage consists of building a bounding volume hierarchy tree for HBV, and involves filling a voxel array with object ID's for USS.

Lastly, the RLE structures for the output image are allocated and initialized. The antialiasing routines in ANT may also require some initialization procedures. In the case of adaptive pixel subdivision, the colors for pixels in the bottom row and leftmost column are precalculated. At this point all structures have been initialized

and the main scanline/pixel/sample loops can be executed (lines 3 through 5 in Figure 5.1).

5.2.2 Shooting Rays

Three methods of filling ray structures are used by the application. These are all found in the RAY library. Top level rays which originate from the eye are determined using the camera transformation and image resolution data stored in the view volume structure. Based on the viewing plane pixel location and any offset within that pixel, first generation rays are calculated. This corresponds to line 6 in Figure 5.1.

When reflected and refracted rays are spawned (lines 13 and 16, respectively), functions are called with appropriate parameters, and the next generation ray is determined. These parameters include the incoming ray (from the previous generation), the surface intersection point, the surface normal at that point, and the reflection or transmission attenuation factor. Also, for the refraction macro, the indices of refraction are extracted from the ATTR structures for the medium the ray traveled from, and/or the medium the ray was entering.

5.2.3 Intersecting Rays with the Environment

The environment intersection method selected in the setup file (SRT, HBV, or USS) is used here to determine the closest intersection of a ray with the objects in the environment (see line 7 in Figure 5.1). The interface to all three of these libraries is identical. A ray and maximum distance are passed to a routine which finds the closest intersection. If there is a successful intersection, several parameter

values are returned. These include the ID of the intersected object, the distance along the ray to the intersection point, the surface point of the intersection (in world coordinates), and the surface normal at this point.

Light ray intersections, used in shadow testing (line 10), are treated slightly differently. In order to determine the intensity at a given surface point, it must be determined, for each individual light, whether the point is illuminated. A *for loop* macro is provided by each environment intersection library to find any objects which block the path of light from a given light source. The macro is called once for each light. A ray is constructed, which originates at the intersection point and is directed at the light, and is passed to the macro. On each loop iteration, the macro sets the object ID parameter to the ID of an object intersected by the ray. This will be some arbitrary object between the ray origin and the light's location. If this object is opaque, then the light does not illuminate the point on the object from which the ray was fired, and the application breaks out of the loop. If, however, the pierced object has some transparency, the light is filtered by the color of that object, and iteration continues.

5.2.4 Texturing and Shading

In determining the contribution from a single light to the intensity at a given point on an object, routines from several libraries are used (see line 11 in Figure 5.1). In order to perform the shading calculations, the reflectance of the surface at that point is needed. If the surface has no texture map, then the reflectance attributes are extracted from the ATTR structure. However, if there is a texture map as-

sociated with this surface, the reflectance at the given point must be determined based on the corresponding texture map location. An inverse mapping routine in the TXM library is called to calculate the parametric coordinates for that point given its object type. These values are then passed with the texture ID number to a TXS routine which performs the sampling of the texture and returns the reflectance.

To compute the spatial and spectral characteristics of the reflected light, a call must be made to a shading routine in the SHAD library for each light source which casts light on the given point. The parameters needed by the shading routine include the shading model being used, the view direction, the surface normal, the surface attributes (possible including the reflectance determined by the texture mapping routines), the light source direction, and the light emission properties. The routine will return an intensity which is then added to any contributions from the other light sources, as well as the separately computed reflected and refracted components.

5.2.5 Antialiasing

Two kinds of antialiasing are supported in this application. The ANT library contains routines for adaptive pixel subdivision and stochastic sampling. If antialiasing is desired for an image, either one may be used. Each has an initialization call which allocates and initializes data structures needed for the given method, and a loop macro which calculates rays used to sample the image plane.

The underlying techniques used to determine final pixel intensities differ sig-

nificantly between the two methods. In the case of adaptive pixel subdivision, intensities for samples along the bottom row and leftmost column are precalculated during initialization. Macros are provided to loop through scanlines, pixels, and samples (lines 3 through 5 in Figure 5.1). For each pixel, a sample loop macro is called with a ray parameter, a color parameter, the view frustum structure, and the current scanline and pixel numbers. On each loop iteration, it is determined by the macro whether enough samples have been taken for the current pixel, based on the differences between samples taken up to that point. If more samples are needed, the ray parameter is filled with a ray shooting through the needed sample point on the pixel area. The intensity for that point is then calculated, and the loop macro subsequently determines whether more rays need to be shot for the current pixel.

The interface for antialiasing using jittering is similar. The sample macro operates in the same fashion, however there are no macros to loop through scanlines and pixels. This is because data from adjacent pixels is not needed for this method, and thus the color calculations for the pixels can be performed in any arbitrary order. This makes the jittering method more appropriate for use in a distributed environment.

5.2.6 Image Output

The images generated by this program are stored in the run length encoded format. The RLE library provides the necessary interface. Initialization routines are called to allocate pixel and scanline data structures, to fill up the RLE image

header given the image resolution, and to open the file (lines 1 and 2). Once these initializations are complete, a routine is called to store individual pixel values. In the normal sequential mode used by this application, where pixels and scanlines are calculated in order, pixels are stored in a single scanline buffer, and written to file automatically when the current scanline is complete (line 20). In a different mode, when pixels are calculated in an arbitrary order, routines which explicitly store pixels or scanlines are used.

5.2.7 Parallelization

A version of this ray tracer has been built which distributes the calculations over a multiprocessor network [TAMP89]. Each node reads in all pertinent files, and keeps a complete copy of all environment data. The program running on these nodes, which calculates pixels or scanlines, was written to interface to the ray tracing routines at a high level to allow parallelization. The interface consists of two calls: an initialization call which reads in all data files and perform initializations, and a call which determines the intensity of a given pixel.

A scheduler process distributes pixels or scanlines to chosen nodes, while a collector process assembles the completed pixels or scanlines for display or storage.

5.3 Results

The following images were rendered with the ray tracing application described in this chapter. To demonstrate the advantages gained by using the MID format, environments from two different sources were rendered. Each source produced an

environment in the MID format, along with an ATTR file describing the associated attributes.

Figure 5.2 shows a ray traced image of the interior of a Pavilion, and Figure 5.3 shows the exterior of the same model. Figure 5.4 shows the interior of a casino, and was modeled with a different modeler from the previous two images.

All three images were rendered using the hierarchical bounding volume acceleration technique from the HBV library. The process of changing the ray tracer to use one of the other environment intersection techniques (either USS or SRT) requires changes to only five lines in the source code and linking to the given library. This allows rapid comparisons between the performance of different techniques to be made.

Figure 5.2: Pavilion interior

Figure 5.3: Pavilion exterior

Figure 5.4: Casino

Chapter 6

Conclusion

A Modular Testbed for Realistic Image Synthesis has been described. The function of this Testbed is to aid in the process of graphics research, by providing researchers with tools enabling them to efficiently construct experimental rendering applications.

This Testbed, implemented at Cornell University's Program of Computer Graphics, differs from other testbeds (including the previous one at Cornell) in several ways. The Testbed is not a production system, as the research focus is on the rendering process itself, requiring flexible, modular tools. Generality and flexibility are provided sometimes at the expense of program data space and execution speed. Also, the building blocks in this Testbed are software libraries bound into the application at link time, as opposed to a collection of filters interconnected by UNIX pipes, a technique used in other testbeds.

The modeling process and rendering process are isolated from each other by an intermediate file format called the Modeler Independent Description (MID). All

modelers within the laboratory will write into this format and all renderers will read from it, enabling each renderer to process environments from any modeler. This isolation also allows the rendering development to advance independently of the modeling development, ensuring compatibility now and in the future.

A three-level structure of library modules is provided. These libraries contain routines commonly needed for building rendering applications, especially those based on ray tracing or radiosity methods. The lowest level of the Testbed contains Utility libraries, which provide basic functionality such as the creation and manipulation of data structures for environments, attributes, and raster images. The middle level contains Object libraries which each perform specific functions on all classes of MID objects. These functions include ray intersection and normal calculations, bounding volume creation and intersection, and conversion between different types of object representations. The highest level of the Testbed library structure is the Rendering level, which provides simple interfaces to complex rendering techniques. This includes shading, texturing, radiosity calculations, and the use of hierarchical structures to accelerate the ray tracing process. Routines from all levels are used by researchers to efficiently construct rendering applications.

A ray tracing application was implemented using Testbed library routines. This program served to demonstrate Testbed usage, as well as to test the various libraries which were used. The mainline code is relatively short and consists largely of calls to Testbed library routines, which demonstrates the power of the Testbed libraries. However, it could be argued that the example is contrived, since it is a simple ray tracing application designed specifically to test library routines. Be-

cause the ray tracing libraries and the ray tracer itself were implemented together, it is conceivable that although the library interfaces were designed to be as general as possible, their design may have been slightly biased towards the ray tracing application.

Two other implementors besides the author have written major applications to exercise the Testbed. Another ray tracer, which had a slightly different functionality, was written, as was a Monte Carlo ray tracer. In general, the libraries were found to be sufficiently powerful to perform the vast majority of the functions required by these applications.

The interfaces to the libraries were found to be clear and flexible, resulting in fairly short mainline programs. Since interfaces are provided at a variety of levels, the implementor is able to use as powerful or as specific a part of a given library as is desired. Specialized functionality is found at the lowest level libraries, while powerful routines making use of several lower and middle level libraries can be found at the high level. Also, different interfaces are provided within a given level, to allow maximum flexibility.

The performance to date of applications written using Testbed routines is not extremely fast. Optimal execution speed was not a goal of the Testbed. The versatile building blocks allow rapid construction of experimental renderers. The flexibility of the interfaces was provided at the expense of some additional overhead. Also, the code is not optimized as it might be for a specialized application. Once techniques have been tested and proven using the Testbed they can be rewritten in an optimized fashion. This is not within the current scope of the testbed.

As a result of the Testbed's modularity, it is straight forward to measure the performance of a certain portion of the computational process. The particular function whose performance is being measured is usually isolated in a clearly defined function call. There is, however, no existing built-in mechanism to aid in the process of performing this statistical monitoring, but this can be added.

Scenes of extremely high complexity can be rendered using Testbed library routines. Because the size of an environment may be so large that it cannot be contained in the memory of a single processor, it is necessary to have the capability of reading a subsection of an environment in order to perform incremental processing. This feature is supported by the MID library.

Some research in parallelism has already been conducted using Testbed routines. However, these routines were used to construct the rendering applications which ran on individual nodes, and did not pertain directly to the parallelism. There are no tools within the present form of the Testbed which directly aid in the parallelization of graphics applications.

There still remains a significant amount of work to be done in the development of the Testbed. At the time of this writing, not all libraries have been completed. For example several radiosity libraries are unfinished. As advances are made in computer graphics hardware, more of the functions in the Testbed will be implemented in hardware.

The applications written using the Testbed to date are heavily based on ray tracing methods. It remains to be seen how the Testbed will assist developers of new techniques which are unrelated to ray tracing or radiosity.

Most of the goals of the Testbed have been achieved. The Testbed provides an environment for experimenting with new light reflection models and global illumination algorithms. The components of a complete rendering system exist, including some simple light reflection models. The researcher can very simply substitute experimental models for these in order to perform comparisons.

Program development time should be dramatically reduced for applications which rely heavily on ray tracing or radiosity. Research applications will be easily maintained because a relatively small portion of the code is written by the developer.

In summary, the Testbed should be a valuable tool for the development of experimental rendering applications. As new image synthesis algorithms are devised, the flexibility of the Testbed should provide a solid platform for future development.

Bibliography

[APPE68] Appel, A. "Some Techniques for Shading Machine Renderings of Solids," in In AFIPS Spring Joint Conference (Atlantic City, New Jersey, April 30 – May 2, 1968, AFIPS, April 1968, pages 37–45.

[ARVO87] Arvo, J. and D. Kirk. "Fast Ray Tracing by Ray Classification," Proceedings of SIGGRAPH'87 (Anaheim, California, July 27–31, 1987), in *Computer Graphics*, 21(4), July 1987, pages 55–64.

[ATHE78] Atherton, P., K. Weiler, and D. Greenberg. "Polygon Shadow Generation," Proceedings of SIGGRAPH'78 (Atlanta, Georgia, August 23–25, 1978), in *Computer Graphics*, 12(3), August 1978, pages 275–281.

[BISH86] Bishop, G. and D. M. Weimer. "Fast Phong Shading," Proceedings of SIGGRAPH'86 (Dallas, Texas, August 18–22, 1986), in *Computer Graphics*, 20(4), August 1986, pages 103–106.

[BLIN77] Blinn, J. F. "Models of Light Reflection for Computer Synthesized Pictures," Proceedings of SIGGRAPH'77 (1977), in *Computer Graphics*, 11(3), Fall 1977, pages 192–198.

[CLEA87] Cleary, J. G. and G. Wyvill. *An Analysis of an Algorithm for Fast Ray-Tracing using Uniform Space Subdivision*, Research Report 87/264/12, Department of Computer Science, University of Calgary, Canada, 1987.

[COHE85a] Cohen, M. F. *A Radiosity Method for the Realistic Image Synthesis of Complex Diffuse Environments*, Master's thesis, Program of Computer Graphics Lab, Cornell University, Ithaca, NY, August 1985.

[COHE85b] Cohen, M. F. and D. P. Greenberg. "The Hemi-Cube: A Radiosity Solution for Complex Environments," Proceedings of SIGGRAPH'85

(San Francisco, California, July 22–26, 1985), in *Computer Graphics*, 19(3), July 1985, pages 31–40.

[COHE88] Cohen, M. F., S. E. Chen, J. R. Wallace, and D. P. Greenberg. "A Progressive Refinement Approach to Fast Radiosity Image Generation," Proceedings of SIGGRAPH'88 (Atlanta, Georgia, August 1–5, 1988), in *Computer Graphics*, August 1988, pages 75–84.

[COOK81] Cook, R. L. and K. E. Torrance. "A Reflectance Model for Computer Graphics," Proceedings of SIGGRAPH'81 (Dallas, Texas, August 3–7, 1981), in *Computer Graphics*, 15(3), August 1981, pages 307–316.

[COOK84] Cook, R. L., T. Porter, and L. Carpenter. "Distributed Ray Tracing," Proceedings of SIGGRAPH'84 (Minneapolis, Minnesota, July 23–27, 1984), in *Computer Graphics*, 18(3), July 1984, pages 137–145.

[COOK86] Cook, R. L. "Stochastic Sampling in Computer Graphics," Developments in Ray Tracing (SIGGRAPH'86 course notes, Dallas, Texas, Aug, 1987), August 1986.

[COOK87] Cook, R. L., L. Carpenter, and E. Catmull. "The Reyes Image Rendering Architecture," Proceedings of SIGGRAPH'87 (Anaheim, California, July 27–31, 1987), in *Computer Graphics*, 21(4), July 1987, pages 95–102.

[CROW78] Crow, F. C. "Shadow Algorithms for Computer Graphics," Proceedings of SIGGRAPH'77 (1977), in *Computer Graphics*, 11(2), Summer 1978, pages 242–248.

[DIPP84] Dippe, M. E. and J. Swensen. "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis," *Proceedings of SIGGRAPH'84 (Minneapolis, Minnesota, July 23–27, 1984)*, 18(3), July 1984, pages 149–158.

[FUJI85] Fujimoto, A. "Accelerated Ray Tracing," in *Computer Graphics: Visual Technology and Art*, Springer Verlag, Tokyo, 1985, pages 41–65.

[FUJI86] Fujimoto, A., T. Takayuki, and I. Kansei. "ARTS: Accelerated Ray-Tracing System," *IEEE Computer Graphics and Applications*, 6(4), April 1986, pages 16–26.

[GLAS84] Glassner, A. S. "Space Subdivision for Fast Ray Tracing," *IEEE Computer Graphics and Applications*, 4(10), October 1984, pages 15–22.

[GOLD71] Goldstein, R. A. and R. Nagel. "3-D Visual Simulation," *Simulation*, 19, January 1971, pages 25–31.

[GOLD87] Goldsmith, J. and J. Salmon. "Automatic Creation of Object Hierarchies for Ray Tracing," *IEEE Computer Graphics and Applications*, 7(5), May 1987, pages 14–20.

[GORA84] Goral, C. M., K. E. Torrence, and D. P. Greenberg. "Modeling the Interaction of Light Between Diffuse Surfaces," Proceedings of SIGGRAPH'84 (Minneapolis, Minnesota, July 23–27, 1984), in *Computer Graphics*, 18(3), July 1984, pages 213–222.

[GORA85] Goral, C. M. *A Model for the Interaction of Light Between Diffuse Surfaces*, Master's thesis, Program of Computer Graphics Lab, Cornell University, Ithaca, NY, January 1985.

[GOUR71] Gouraud, H. "Continuous Shading of Curved Surfaces," *IEEE Transactions on Computers*, C-20(6), June 1971, pages 623–628.

[HAIN86] Haines, E. A. and D. P. Greenberg. "The Light buffer: a Shadow Testing Accelerator," *IEEE Computer Graphics and Applications*, 6(9), September 1986, pages 6–16.

[HALL83] Hall, R. A. and D. P. Greenberg. "A Testbed for Realistic Image Synthesis," *IEEE Computer Graphics and Applications*, 3(8), November 1983, pages 10–20.

[HOWE87] Howe, C. D. and B. Moxon. "How to Program Parallel Processors," *IEEE Spectrum*, 24(9), September 1987, pages 36–41.

[IMME86] Immel, D. S., M. F. Cohen, and D. P. Greenberg. "A Radiosity Method for Non-Diffuse Environments," Proceedings of SIGGRAPH'86 (Dallas, Texas, August 18–22, 1986), in *Computer Graphics*, 20(4), August 1986, pages 133–142.

[KAJI86] Kajiya, J. T. "The Rendering Equation," Proceedings of SIGGRAPH'86 (Dallas, Texas, August 18–22, 1986), in *Computer Graphics*, 20(4), August 1986, pages 143–150.

[KAPL85] Kaplan, M. R. "Space-Tracing, A Constant Time Ray-Tracer," SIGGRAPH'85 State of the Art in Image Synthesis seminar notes, July 1985.

[KAY86] Kay, T. L. and J. T. Kajiya. "Ray Tracing Complex Scenes," Proceedings of SIGGRAPH'86 (Dallas, Texas, August 18–22, 1986), in *Computer Graphics*, 20(4), August 1986, pages 269–278.

[KEDE84] Keden, G. and J. L. Ellis. "The Raycasting Machine," in Proceedings of ICCD'84, IEEE Computer Society Press, Silver Spring, MD, October 1984, pages 533–538.

[KOBA87] Kobayashi, H., T. Nakamura, and S. Yoshiharu. "Parallel Processing of an Object Space for Image Synthesis Using Ray Tracing," *The Visual Computer*, 3(1), February 1987, pages 13–22.

[MEYE83] Meyer, G. W. *Colorimetry and Computer Graphics*, PhD dissertation, Cornell University, January 1983.

[MEYE86] Meyer, G. W. *Color Calculations for and Perceptual Assessment of Computer Grahpic Images*, PhD dissertation, Cornell University, June 1986.

[MURA83] Murakami, K. and H. Matsumoto. "Ray Tracing with Octree Data Structures," in Proceedings of the 28th Information Processing Conference, 1983.

[NADA87] Nadas, T. and A. Fournier. "GRAPE: An Environment to Build Display Processes," Proceedings of SIGGRAPH'87 (Anaheim, California, July 27–31, 1987), in *Computer Graphics*, 21(4), July 1987, pages 75–84.

[NEMO86] Nemoto, K. and O. Takao. "An Adaptive Subdivision by Sliding Boundary Surfaces for Fast Ray Tracing," in Proceedings of Graphics Interface'86, May 1986, pages 43–48.

[PETE86] Peterson, J. W., R. G. Bogart, and S. W. Thomas. *The Utah Raster Toolkit*, Technical Report, University of Utah, 1986.

[PHON75] Phong, B. "Illumination for Computer-Generated Pictures," *Communications of the ACM*, 18(6), June 1975, pages 311–317.

[POTM87] Potmesil, M. "FRAMES: Software Tools for Modeling, Rendering and Animation of 3D Scenes," Proceedings of SIGGRAPH'87 (Anaheim, California, July 27–31, 1987), in *Computer Graphics*, 21(4), July 1987, pages 85–93.

[ROTH82] Roth, S. D. "Ray Casting for Modeling Solids," *Computer Graphics and Image Processing*, 18(2), February 1982, pages 109–144.

[RUBI80] Rubin, S. M. and T. Whitted. "A Three-Dimensional Representation for Fast Rendering of Complex Scenes," Proceedings of SIGGRAPH'80 (1980), in *Computer Graphics*, July 1980, pages 110–116.

[RUSH87] Rushmeier, H. E. and K. E. Torrance. "The Zonal Method for Calculating Light Intensities in the Presence of a Participating Mediuim," Proceedings of SIGGRAPH'87 (Anaheim, California, July 27–31, 1987), in *Computer Graphics*, 21(4), July 1987, pages 293–302.

[RUSH88] Rushmeier, H. E. *Realistic Image Synthesis for Scenes with Radiatively Participating Media*, PhD dissertation, Cornell University, June 1988.

[SAME84] Samet, H. "The Quadtree and Related Hierarchical Data Structures," *ACM Computing Surveys*, 16(2), June 1984, pages 187–260.

[SATA85] Sata, H., M. Ishii, K. Sato, M. Ikesaka, H. Ishihata, M. Kakimoto, K. Hirota, and K. Inoue. "Fast Image Generation of Constructive Solid Geometry Using A Cellular Array Processor," Proceedings of SIGGRAPH'85 (San Francisco, California, July 22–26, 1985), in *Computer Graphics*, 19(3), July 1985, pages 95–102.

[SCHE87] Scherson, I. D. and E. Caspary. "Data Structures and the Time Complexity of Ray Tracing," *The Visual Computer*, 3(4), December 1987, pages 201–213.

[SUTH74] Sutherland, I. E., R. F. Sproull, and R. A. Schumacker. "A Characterization of Ten Hidden-Surface Algorithms," *Computing Surveys*, 6(1), March 1974, pages 1–55.

[TAMP89] Tampieri, F. *Global Illumination Algorithms for Parallel Computer Architectures*, Master's thesis, Program of Computer Graphics, Cornell University, Ithaca, NY, August 1989.

[THOM86] Thomas, S. W. *Design of the Utah RLE Format*, Technical Report 86-15, University of Utah, November 1986.

[TORR67] Torrance, K. E. and E. M. Sparrow. "Theory for Off-Specular Reflection from Roughened Surfaces," *J. Opt. Soc. Am.*, 57(9), May 1967, pages 1105–1114.

[WALL87] Wallace, J. R., M. F. Cohen, and D. P. Greenberg. "A Two-Pass Solution to the Rendering Equation: A Synthesis of Ray Tracing and Radiosity Methods," Proceedings of SIGGRAPH'87 (Anaheim, California, July 27–31, 1987), in *Computer Graphics*, 21(4), July 1987, pages 311–320.

[WALL88] Wallace, J. R. *A Two-Pass Solution to the Rendering Equation: A Synthesis of Ray Tracing and Radiosity Methods*, Master's thesis, Program of Computer Graphics Lab, Cornell University, Ithaca, NY, January 1988.

[WARD88] Ward, G. J., F. M. Rubinstein, and C. R. D. "A Ray Tracing Solution for Diffuse Interreflection," Proceedings of SIGGRAPH'88 (Atlanta, Georgia, August 1–5, 1988), in *Computer Graphics*, August 1988, pages 85–92.

[WARN69] Warnock, J. E. *A Hidden Surface Algorithm for Computer Generated Halftone Pictures*, Technical Report 4-8, University of Utah, June 1969.

[WATK70] Watkins, G. S. *A Real-Time Visible Surface Algorithm*, Technical Report UTEC-CSc-70-101, University of Utah, June 1970.

[WEGH84] Weghorst, H., G. Hooper, and D. P. Greenberg. "Improved Computational Method for Ray Tracing," *ACM Transactions on Graphics*, 3(1), January 1984, pages 52–69.

[WEIL86] Weiler, K. J. *Topological Structures for Geometric Modeling*, PhD dissertation, Rensselaer Polytechnic Institute, August 1986.

[WEIN81] Weinberg, R. "Parallel Processing Image Synthesis and Anti-Aliasing," Proceedings of SIGGRAPH'81 (Dallas, Texas, August 3–7, 1981), in *Computer Graphics*, 15(3), August 1981, pages 55–62.

[WHIT80] Whitted, T. "An Improved Illumination Model for Shaded Display," *Communications of the ACM*, 23(6), June 1980, pages 343–349.

[WHIT82] Whitted, T. and D. M. Weimer. "A Software Testbed for the Development of 3D Raster Graphics Systems," *ACM Transactions on Graphics*, 1, January 1982, pages 43–58.